RTI Connext C++ API

Version 5.0.0

Generated by Doxygen 1.5.5

Mon Aug 13 09:00:30 2012

Contents

Chapter 1

RTI Connext

Core Libraries and Utilities

Real-Time Innovations, Inc.

RTI Connext is network middleware for real-time distributed applications. It provides the communications services that programmers need to distribute time-critical data between embedded and/or enterprise devices or nodes. RTI Connext uses the publish-subscribe communications model to make data distribution efficient and robust.

The RTI Connext Application Programming Interface (API) is based on the OMG's Data Distribution Service (DDS) specification. The most recent publication of this specification can be found in the Catalog of OMG Specifications under "Middleware Specifications".

1.1 Feedback and Support for this Release.

For more information, visit our knowledge base (accessible from https://support.rti.com/) to see sample code, general information on RTI Connext, performance information, troubleshooting tips, and technical details.

By its very nature, the knowledge base is continuously evolving and improving. We hope that you will find it helpful. If there are questions that you would like to see addressed or comments you would like to share, please send e-mail to support@rti.com. We can only guarantee a response for customers with a current maintenance contract or subscription. To purchase a maintenance contract or subscription, contact your local RTI representative

(see `http://www.rti.com/company/contact.html`), send an email request to `sales@rti.com`, or call +1 (408) 990-7400.

Please do not hesitate to contact RTI with questions or comments about this release. We welcome any input on how to improve RTI Connext to suit your needs.

1.2 Available Documentation.

The documentation for this release is provided in two forms: the HTML API reference documentation and PDF documents. If you are new to RTI Connext, the **Documentation Roadmap** (p. 197) will provide direction on how to learn about this product.

1.2.1 The PDF documents for the Core Libraries and Utilities are:

- `What's New.` An overview of the new features in this release.
- `Release Notes.` System requirements, compatibility, what's fixed in this release, and known issues.
- `Getting Started Guide.` Download and installation instructions. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. Developers should read this document first.
- `Getting Started Guide, Database Addendum.` Additional installation and setup information for database usage.
- `Getting Started Guide, Embedded Systems Addendum.` Additional installation and setup information for embedded systems.
- `User's Manual.` Introduction to RTI Connext, product tour and conceptual presentation of the functionality of RTI Connext.
- `Platform Notes.` Specific details, such as compilation setting and libraries, related to building and using RTI Connext on the various supported platforms.
- `QoS Reference Guide.` A compact summary of supported Quality of Service (QoS) policies.

- `XML-Based Application Creation Getting Started Guide.` Details on how to use XML-Based Application Creation, an experimental feature in this release.

- `C API Reference Manual.` A consolidated PDF version of the HTML C API reference documentation.

- `C++ API Reference Manual.` A consolidated PDF version of the HTML C++ API reference documentation.

- `Java API Reference Manual.` A consolidate PDF version of the HTML Java API reference documentation.

- `.NET API Reference Manual.` A consolidated PDF version of the HTML .Net API reference documentation.

1.2.2 The HTML API Reference documentation contains:

- **DDS API Reference** (p. 203) - The DDS API reference.

- **RTI Connext API Reference** (p. 216) - RTI Connext API's independent of the DDS standard.

- **Programming How-To's** (p. 217) - Describes and shows the common tasks done using the API.

- **Programming Tools** (p. 219) - RTI Connext helper tools.

The HTML API Reference documentation can be accessed through the tree view in the left frame of the web browser window. The bulk of the documentation is found under the entry labeled "Modules".

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

Module Documentation

5.1 Clock Selection

APIs related to clock selection. RTI Connext uses clocks to measure time and generate timestamps.

The middleware uses two clocks, an internal clock and an external clock. The internal clock is used to measure time and handles all timing in the middleware. The external clock is used solely to generate timestamps, such as the source timestamp and the reception timestamp, in addition to providing the time given by **DDSDomainParticipant::get_current_time** (p. 1191).

5.1.1 Available Clocks

Two clock implementations are generally available, the monotonic clock and the realtime clock.

The monotonic clock provides times that are monotonic from a clock that is not adjustable. This clock is useful to use in order to not be subject to changes in the system or realtime clock, which may be adjusted by the user or via time synchronization protocols. However, this time generally starts from an arbitrary point in time, such as system startup. Note that this clock is not available for all architectures. Please see the `Platform Notes` for the architectures on which it is supported. For the purposes of clock selection, this clock can be referenced by the name "monotonic".

The realtime clock provides the realtime of the system. This clock may generally be monotonic but may not be guaranteed to be so. It is adjustable and may be subject to small and large changes in time. The time obtained from this clock is generally a meaningful time in that it is the amount of time from a known

epoch. For the purposes of clock selection, this clock can be referenced by the names "realtime" or "system".

5.1.2 Clock Selection Strategy

By default, both the internal and external clocks use the real-time clock. If you want your application to be robust to changes in the system time, you may use the monotonic clock as the internal clock, and leave the system clock as the external clock. Note, however, that this may slightly diminish performance in that both the send and receive paths may need to obtain times from both clocks. Since the monotonic clock is not available on all architectures, you may want to specify "monotonic,realtime" for the internal_clock (see the table below). By doing so, the middleware will attempt to use the monotonic clock if available, and will fall back to the realtime clock if the monotonic clock is not available.

If you want your application to be robust to changes in the system time, you are not relying on source timestamps, and you want to avoid obtaining times from both clocks, you may use the monotonic clock for both the internal and external clocks.

5.1.3 Configuring Clock Selection

To configure the clock selection, use the **PROPERTY** (p. 436) QoS policy associated with the **DDSDomainParticipant** (p. 1139).

See also:

DDS_PropertyQosPolicy (p. 834)

The following table lists the supported clock selection properties.

Property	Description
dds.clock.external_clock	Comma-delimited list of clocks to use for the external clock, in the order of preference. Valid clock names are "realtime", "system", and "monotonic". Default: "realtime"
dds.clock.internal_clock	Comma-delimited list of clocks to use for the internal clock, in the order of preference. Valid clock names are "realtime", "system", and "monotonic". Default: "realtime"

Table 5.1: *Clock Selection Properties*

5.2 Domain Module

Contains the **DDSDomainParticipant** (p. 1139) class that acts as an entry-point of RTI Connext and acts as a factory for many of the classes. The **DDSDomainParticipant** (p. 1139) also acts as a container for the other objects that make up RTI Connext.

Modules

- **DomainParticipantFactory**

  DDSDomainParticipantFactory *(p. 1216) entity and associated elements*

- **DomainParticipants**

  DDSDomainParticipant *(p. 1139) entity and associated elements*

- **Built-in Topics**

  Built-in objects created by RTI Connext but accessible to the application.

Defines

- #define **DDSTheParticipantFactory** DDSDomainParticipantFactory::get_instance()

  Can be used as an alias for the singleton factory returned by the operation ***DDSDomainParticipantFactory::get_instance()*** *(p. 1221).*

Typedefs

- typedef DDS_DOMAINID_TYPE_NATIVE **DDS_DomainId_t**

  An integer that indicates in which domain a ***DDSDomainParticipant*** *(p. 1139) communicates.*

5.2.1 Detailed Description

Contains the **DDSDomainParticipant** (p. 1139) class that acts as an entry-point of RTI Connext and acts as a factory for many of the classes. The **DDSDomainParticipant** (p. 1139) also acts as a container for the other objects that make up RTI Connext.

5.2.2 Define Documentation

5.2.2.1 #define DDSTheParticipantFactory DDSDomainParticipantFactory::get_instance()

Can be used as an alias for the singleton factory returned by the operation **DDSDomainParticipantFactory::get_instance()** (p. 1221).

See also:

DDSDomainParticipantFactory::get_instance (p. 1221)

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

5.2.3 Typedef Documentation

5.2.3.1 typedef DDS_DOMAINID_TYPE_NATIVE DDS_DomainId_t

An integer that indicates in which domain a **DDSDomainParticipant** (p. 1139) communicates.

Participants with the same **DDS_DomainId_t** (p. 33) are said to be in the same domain, and can thus communicate with one another.

The lower limit for a domain ID is 0. The upper limit for a domain ID is determined by the guidelines stated in **DDS_RtpsWellKnownPorts_t** (p. 905) (specifically **DDS_RtpsWellKnownPorts_t::domain_id_gain** (p. 907))

5.3 DomainParticipantFactory

DDSDomainParticipantFactory (p. 1216) entity and associated elements

Classes

- class **DDSDomainParticipantFactory**

  <<**singleton**>> (p. 200) <<**interface**>> (p. 199) *Allows creation and destruction of* ***DDSDomainParticipant*** *(p. 1139) objects.*

- struct **DDS_DomainParticipantFactoryQos**

  QoS policies supported by a ***DDSDomainParticipantFactory*** *(p. 1216).*

Typedefs

- typedef **DDS_ReturnCode_t**(∗ **DDSDomainParticipantFactory_RegisterTypeFunction**)(**DDSDomainParticipant** ∗participant, const char ∗type_name)

  Prototype of a register type function.

Variables

- struct **DDS_DomainParticipantQos DDS_PARTICIPANT_QOS_DEFAULT**

  Special value for creating a DomainParticipant with default QoS.

5.3.1 Detailed Description

DDSDomainParticipantFactory (p. 1216) entity and associated elements

5.3.2 Typedef Documentation

5.3.2.1 typedef DDS_ReturnCode_t(∗ DDSDomainParticipantFactory_RegisterTypeFunction)(DDSDomainParticipant ∗participant, const char ∗type_name)

Prototype of a register type function.

Parameters:

participant *<<inout>>* *(p. 200)* **DDSDomainParticipant** (p. 1139) participant the type is registered with.

type_name *<<in>>* *(p. 200)* Name the type is registered with.

Returns:

One of the **Standard Return Codes** (p. 314)

5.3.3 Variable Documentation

5.3.3.1 struct DDS_DomainParticipantQos DDS_PARTICIPANT_QOS_DEFAULT

Special value for creating a DomainParticipant with default QoS.

When used in **DDSDomainParticipantFactory::create_participant** (p. 1233), this special value is used to indicate that the **DDSDomainParticipant** (p. 1139) should be created with the default **DDSDomainParticipant** (p. 1139) QoS by means of the operation **DDSDomainParticipantFactory::get_default_participant_qos()** (p. 1224) and using the resulting QoS to create the **DDSDomainParticipant** (p. 1139).

When used in **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222) operation had never been called.

When used in **DDSDomainParticipant::set_qos** (p. 1197), this special value is used to indicate that the QoS of the **DDSDomainParticipant** (p. 1139) should be changed to match the current default QoS set in the **DDSDomainParticipantFactory** (p. 1216) that the **DDSDomainParticipant** (p. 1139) belongs to.

RTI Connext treats this special value as a constant.

Note: You cannot use this value to *get* the default QoS values from the DomainParticipant factory; for this purpose, use **DDSDomainParticipantFactory::get_default_participant_qos** (p. 1224).

See also:

NDDS_DISCOVERY_PEERS (p. 388)
DDSDomainParticipantFactory::create_participant() (p. 1233)
DDSDomainParticipantFactory::set_default_participant_qos() (p. 1222)
DDSDomainParticipant::set_qos() (p. 1197)

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

5.4 DomainParticipants

DDSDomainParticipant (p. 1139) entity and associated elements

Classes

- class **DDSDomainParticipantListener**

  <<**interface**>> (p. 199) *Listener for participant status.*

- class **DDSDomainParticipant**

  <<**interface**>> (p. 199) *Container for all* ***DDSDomainEntity*** *(p. 1138) objects.*

- struct **DDS_DomainParticipantQos**

  QoS policies supported by a ***DDSDomainParticipant*** *(p. 1139) entity.*

Variables

- struct **DDS_TopicQos DDS_TOPIC_QOS_DEFAULT**

  Special value for creating a ***DDSTopic*** *(p. 1419) with default QoS.*

- struct **DDS_PublisherQos DDS_PUBLISHER_QOS_DEFAULT**

  Special value for creating a ***DDSPublisher*** *(p. 1346) with default QoS.*

- struct **DDS_SubscriberQos DDS_SUBSCRIBER_QOS_DEFAULT**

  Special value for creating a ***DDSSubscriber*** *(p. 1390) with default QoS.*

- struct **DDS_FlowControllerProperty_t DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT**

  <<**eXtension**>> (p. 199) *Special value for creating a* ***DDSFlowController*** *(p. 1259) with default property.*

- const char *const **DDS_SQLFILTER_NAME**

  <<**eXtension**>> (p. 199) *The name of the built-in SQL filter that can be used with ContentFilteredTopics and MultiChannel DataWriters.*

- const char *const **DDS_STRINGMATCHFILTER_NAME**

  <<**eXtension**>> (p. 199) *The name of the built-in StringMatch filter that can be used with ContentFilteredTopics and MultiChannel DataWriters.*

5.4.1 Detailed Description

DDSDomainParticipant (p. 1139) entity and associated elements

5.4.2 Variable Documentation

5.4.2.1 struct DDS_TopicQos DDS_TOPIC_QOS_DEFAULT

Special value for creating a **DDSTopic** (p. 1419) with default QoS.

When used in **DDSDomainParticipant::create_topic** (p. 1175), this special value is used to indicate that the **DDSTopic** (p. 1419) should be created with the default **DDSTopic** (p. 1419) QoS by means of the operation get_default_topic_qos and using the resulting QoS to create the **DDSTopic** (p. 1419).

When used in **DDSDomainParticipant::set_default_topic_qos** (p. 1162), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSDomainParticipant::set_default_topic_qos** (p. 1162) operation had never been called.

When used in **DDSTopic::set_qos** (p. 1421), this special value is used to indicate that the QoS of the **DDSTopic** (p. 1419) should be changed to match the current default QoS set in the **DDSDomainParticipant** (p. 1139) that the **DDSTopic** (p. 1419) belongs to.

Note: You cannot use this value to *get* the default QoS values for a Topic; for this purpose, use **DDSDomainParticipant::get_default_topic_qos** (p. 1161).

See also:

DDSDomainParticipant::create_topic (p. 1175)
DDSDomainParticipant::set_default_topic_qos (p. 1162)
DDSTopic::set_qos (p. 1421)

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

5.4.2.2 struct DDS_PublisherQos DDS_PUBLISHER_QOS_DEFAULT

Special value for creating a **DDSPublisher** (p. 1346) with default QoS.

When used in **DDSDomainParticipant::create_publisher** (p. 1169), this special value is used to indicate that the **DDSPublisher** (p. 1346) should be created with the default **DDSPublisher** (p. 1346) QoS by means of the operation

get_default_publisher_qos and using the resulting QoS to create the **DDSPublisher** (p. 1346).

When used in **DDSDomainParticipant::set_default_publisher_qos** (p. 1164), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSDomainParticipant::set_default_publisher_qos** (p. 1164) operation had never been called.

When used in **DDSPublisher::set_qos** (p. 1366), this special value is used to indicate that the QoS of the **DDSPublisher** (p. 1346) should be changed to match the current default QoS set in the **DDSDomainParticipant** (p. 1139) that the **DDSPublisher** (p. 1346) belongs to.

Note: You cannot use this value to *get* the default QoS values for a Publisher; for this purpose, use **DDSDomainParticipant::get_default_publisher_qos** (p. 1163).

See also:

DDSDomainParticipant::create_publisher (p. 1169)
DDSDomainParticipant::set_default_publisher_qos (p. 1164)
DDSPublisher::set_qos (p. 1366)

Examples:

HelloWorld_publisher.cxx.

5.4.2.3 struct DDS_SubscriberQos DDS_SUBSCRIBER_QOS_DEFAULT

Special value for creating a **DDSSubscriber** (p. 1390) with default QoS.

When used in **DDSDomainParticipant::create_subscriber** (p. 1172), this special value is used to indicate that the **DDSSubscriber** (p. 1390) should be created with the default **DDSSubscriber** (p. 1390) QoS by means of the operation get_default_subscriber_qos and using the resulting QoS to create the **DDSSubscriber** (p. 1390).

When used in **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167) operation had never been called.

When used in **DDSSubscriber::set_qos** (p. 1410), this special value is used to indicate that the QoS of the **DDSSubscriber** (p. 1390) should be changed to match the current default QoS set in the **DDSDomainParticipant** (p. 1139) that the **DDSSubscriber** (p. 1390) belongs to.

Note: You cannot use this value to *get* the default QoS values for a Subscriber; for this purpose, use **DDSDomainParticipant::get_default_subscriber_qos** (p. 1166).

See also:

DDSDomainParticipant::create_subscriber (p. 1172)
DDSDomainParticipant::get_default_subscriber_qos (p. 1166)
DDSSubscriber::set_qos (p. 1410)

Examples:

HelloWorld_subscriber.cxx.

5.4.2.4 struct DDS_FlowControllerProperty_t DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT

<<eXtension>> (p. 199) Special value for creating a **DDSFlowController** (p. 1259) with default property.

When used in **DDSDomainParticipant::create_flowcontroller** (p. 1184), this special value is used to indicate that the **DDSFlowController** (p. 1259) should be created with the default **DDSFlowController** (p. 1259) property by means of the operation get_default_flowcontroller_property and using the resulting QoS to create the **DDS_FlowControllerProperty_t** (p. 749).

When used in **DDSDomainParticipant::set_default_flowcontroller_property** (p. 1155), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSDomainParticipant::set_default_flowcontroller_property** (p. 1155) operation had never been called.

When used in **DDSFlowController::set_property** (p. 1260), this special value is used to indicate that the property of the **DDSFlowController** (p. 1259) should be changed to match the current default property set in the **DDSDomainParticipant** (p. 1139) that the **DDSFlowController** (p. 1259) belongs to.

Note: You cannot use this value to *get* the default properties for a FlowController; for this purpose, use **DDSDomainParticipant::get_default_flowcontroller_property** (p. 1154).

See also:

DDSDomainParticipant::create_flowcontroller (p. 1184)
DDSDomainParticipant::set_default_flowcontroller_property (p. 1155)
DDSFlowController::set_property (p. 1260)

5.4.2.5 const char* const DDS_SQLFILTER_NAME

<<eXtension>> (p. 199) The name of the built-in SQL filter that can be used with ContentFilteredTopics and MultiChannel DataWriters.

See also:

Queries and Filters Syntax (p. 208)

5.4.2.6 const char* const DDS_STRINGMATCHFILTER_NAME

<<eXtension>> (p. 199) The name of the built-in StringMatch filter that can be used with ContentFilteredTopics and MultiChannel DataWriters.

The StringMatch Filter is a subset of the SQL filter; it only supports the MATCH relational operator on a single string field.

See also:

Queries and Filters Syntax (p. 208)

5.5 Built-in Topics

Built-in objects created by RTI Connext but accessible to the application.

Modules

- **Participant Built-in Topics**

  Builtin topic for accessing information about the DomainParticipants discovered by RTI Connext.

- **Topic Built-in Topics**

  Builtin topic for accessing information about the Topics discovered by RTI Connext.

- **Publication Built-in Topics**

  Builtin topic for accessing information about the Publications discovered by RTI Connext.

- **Subscription Built-in Topics**

  Builtin topic for accessing information about the Subscriptions discovered by RTI Connext.

Classes

- struct **DDS_Locator_t**

  <<**eXtension**>> (p. 199) *Type used to represent the addressing information needed to send a message to an RTPS Endpoint using one of the supported transports.*

- struct **DDS_LocatorSeq**

  Declares IDL `sequence` *<* ***DDS_Locator_t*** *(p. 783) >.*

- struct **DDS_ProtocolVersion_t**

  <<**eXtension**>> (p. 199) *Type used to represent the version of the RTPS protocol.*

- struct **DDS_VendorId_t**

  <<**eXtension**>> (p. 199) *Type used to represent the vendor of the service implementing the RTPS protocol.*

- struct **DDS_ProductVersion_t**

<<**eXtension**>> (p. 199) *Type used to represent the current version of RTI Connext.*

ˆ struct **DDS_BuiltinTopicKey_t**

The key type of the built-in topic types.

ˆ struct **DDS_ContentFilterProperty_t**

<<**eXtension**>> (p. 199) *Type used to provide all the required information to enable content filtering.*

Defines

ˆ #define **DDS_LOCATOR_ADDRESS_LENGTH_MAX** 16

Declares length of address field in locator.

ˆ #define **DDS_PROTOCOLVERSION_1_0** { 1, 0 }

The protocol version 1.0.

ˆ #define **DDS_PROTOCOLVERSION_1_1** { 1, 1 }

The protocol version 1.1.

ˆ #define **DDS_PROTOCOLVERSION_1_2** { 1, 2 }

The protocol version 1.2.

ˆ #define **DDS_PROTOCOLVERSION_2_0** { 2, 0 }

The protocol version 2.0.

ˆ #define **DDS_PROTOCOLVERSION_2_1** { 2, 1 }

The protocol version 2.1.

ˆ #define **DDS_PROTOCOLVERSION** { 2, 1 }

The most recent protocol version. Currently 1.2.

ˆ #define **DDS_VENDOR_ID_LENGTH_MAX** 2

Length of vendor id.

ˆ #define **DDS_PRODUCTVERSION_UNKNOWN** { 0, 0, 0, 0 }

The value used when the product version is unknown.

Variables

- struct **DDS_Locator_t DDS_LOCATOR_INVALID**

  An invalid locator.

- const **DDS_Long DDS_LOCATOR_KIND_INVALID**

  Locator of this kind is invalid.

- const **DDS_UnsignedLong DDS_LOCATOR_PORT_INVALID**

  An invalid port.

- const **DDS_Octet DDS_LOCATOR_ADDRESS_INVALID** [DDS_LOCATOR_ADDRESS_LENGTH_MAX]

  An invalid address.

- const **DDS_Long DDS_LOCATOR_KIND_UDPv4**

  A locator for a UDPv4 address.

- const **DDS_Long DDS_LOCATOR_KIND_SHMEM**

  A locator for an address acessed via shared memory.

- const **DDS_Long DDS_LOCATOR_KIND_UDPv6**

  A locator for a UDPv6 address.

- const **DDS_Long DDS_LOCATOR_KIND_RESERVED**

  Locator of this kind is reserved.

5.5.1 Detailed Description

Built-in objects created by RTI Connext but accessible to the application.

RTI Connext must discover and keep track of the remote entities, such as new participants in the domain. This information may also be important to the application, which may want to react to this discovery, or else access it on demand.

A set of built-in topics and corresponding **DDSDataReader** (p. 1087) objects are introduced to be used by the application to access these discovery information.

The information can be accessed as if it was normal application data. This allows the application to know when there are any changes in those values by means of the **DDSListener** (p. 1318) or the **DDSCondition** (p. 1075) mechanisms.

The built-in data-readers all belong to a built-in **DDSSubscriber** (p. 1390), which can be retrieved by using the method **DDSDomainParticipant::get_builtin_subscriber** (p. 1186). The built-in **DDSDataReader** (p. 1087) objects can be retrieved by using the operation **DDSSubscriber::lookup_datareader** (p. 1404), with the topic name as a parameter.

Built-in entities have default listener settings as well. The built-in **DDSSubscriber** (p. 1390) and all of its built-in topics have 'nil' listeners with all statuses appearing in their listener masks (acting as a NO-OP listener that does not reset communication status). The built-in DataReaders have null listeners with no statuses in their masks.

The information that is accessible about the remote entities by means of the built-in topics includes all the QoS policies that apply to the corresponding remote Entity. This QoS policies appear as normal 'data' fields inside the data read by means of the built-in Topic. Additional information is provided to identify the Entity and facilitate the application logic.

The built-in **DDSDataReader** (p. 1087) will not provide data pertaining to entities created from the same **DDSDomainParticipant** (p. 1139) under the assumption that such entities are already known to the application that created them.

Refer to **DDS_ParticipantBuiltinTopicData** (p. 816), **DDS_TopicBuiltinTopicData** (p. 958), **DDS_SubscriptionBuiltinTopicData** (p. 936) and **DDS_PublicationBuiltinTopicData** (p. 839) for a description of all the built-in topics and their contents.

The QoS of the built-in **DDSSubscriber** (p. 1390) and **DDSDataReader** (p. 1087) objects is given by the following table:

5.5.2 Define Documentation

5.5.2.1 #define DDS_LOCATOR_ADDRESS_LENGTH_MAX 16

Declares length of address field in locator.

5.5.2.2 #define DDS_PROTOCOLVERSION_1_0 { 1, 0 }

The protocol version 1.0.

5.5.2.3 #define DDS_PROTOCOLVERSION_1_1 { 1, 1 }

The protocol version 1.1.

5.5.2.4 #define DDS_PROTOCOLVERSION_1_2 { 1, 2 }

The protocol version 1.2.

5.5.2.5 #define DDS_PROTOCOLVERSION_2_0 { 2, 0 }

The protocol version 2.0.

5.5.2.6 #define DDS_PROTOCOLVERSION_2_1 { 2, 1 }

The protocol version 2.1.

5.5.2.7 #define DDS_PROTOCOLVERSION { 2, 1 }

The most recent protocol version. Currently 1.2.

5.5.2.8 #define DDS_VENDOR_ID_LENGTH_MAX 2

Length of vendor id.

5.5.2.9 #define DDS_PRODUCTVERSION_UNKNOWN { 0, 0, 0, 0 }

The value used when the product version is unknown.

5.5.3 Variable Documentation

5.5.3.1 struct DDS_Locator_t DDS_LOCATOR_INVALID

An invalid locator.

5.5.3.2 const DDS_Long DDS_LOCATOR_KIND_INVALID

Locator of this kind is invalid.

5.5.3.3 const DDS_UnsignedLong DDS_LOCATOR_PORT_INVALID

An invalid port.

5.5.3.4 const DDS_Octet DDS_LOCATOR_ADDRESS_INVALID[DDS_LOCATOR_ADDRESS_LENGTH_MAX]

An invalid address.

5.5.3.5 const DDS_Long DDS_LOCATOR_KIND_UDPv4

A locator for a UDPv4 address.

5.5.3.6 const DDS_Long DDS_LOCATOR_KIND_SHMEM

A locator for an address acessed via shared memory.

5.5.3.7 const DDS_Long DDS_LOCATOR_KIND_UDPv6

A locator for a UDPv6 address.

5.5.3.8 const DDS_Long DDS_LOCATOR_KIND_RESERVED

Locator of this kind is reserved.

QoS	Value
DDS_UserDataQosPolicy (p. 1048)	0-length sequence
DDS_TopicDataQosPolicy (p. 963)	0-length sequence
DDS_GroupDataQosPolicy (p. 755)	0-length sequence
DDS_DurabilityQosPolicy (p. 614)	**DDS_TRANSIENT_LOCAL_-DURABILITY_QOS** (p. 349)
DDS_-DurabilityServiceQosPolicy (p. 618)	Does not apply as **DDS_DurabilityQosPolicyKind** (p. 348) is **DDS_TRANSIENT_-LOCAL_DURABILITY_QOS** (p. 349)
DDS_PresentationQosPolicy (p. 823)	access_scope = **DDS_TOPIC_-PRESENTATION_QOS** (p. 352) coherent_access = **DDS_BOOLEAN_FALSE** (p. 299) ordered_access = **DDS_BOOLEAN_FALSE** (p. 299)
DDS_DeadlineQosPolicy (p. 567)	Period = infinite
DDS_LatencyBudgetQosPolicy (p. 771)	duration = 0
DDS_OwnershipQosPolicy (p. 807)	**DDS_SHARED_-OWNERSHIP_QOS** (p. 356)
DDS_-OwnershipStrengthQosPolicy (p. 814)	value = 0
DDS_LivelinessQosPolicy (p. 779)	kind = **DDS_AUTOMATIC_-LIVELINESS_QOS** (p. 359) lease_duration = 0
DDS_-TimeBasedFilterQosPolicy (p. 954)	minimum_separation = 0
DDS_PartitionQosPolicy (p. 820)	0-length sequence
DDS_ReliabilityQosPolicy (p. 865)	kind = **DDS_RELIABLE_-RELIABILITY_QOS** (p. 363) max_blocking_time = 100 milliseconds
DDS_-DestinationOrderQosPolicy (p. 570)	**DDS_BY_RECEPTION_-TIMESTAMP_-DESTINATIONORDER_QOS** (p. 366)
DDS_HistoryQosPolicy (p. 758)	kind = **DDS_KEEP_LAST_-HISTORY_QOS** (p. 368) depth = 1
DDS_ResourceLimitsQosPolicy (p. 879)	max_samples = **DDS_LENGTH_UNLIMITED** (p. 371) max_instances = **DDS_LENGTH_UNLIMITED** (p. 371) max_samples_per_instance = **DDS_LENGTH_UNLIMITED**

5.6 Topic Module

Contains the **DDSTopic** (p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322) classes, the **DDSTopicListener** (p. 1430) interface, and more generally, all that is needed by an application to define **DDSTopic** (p. 1419) objects and attach QoS policies to them.

Modules

- **Topics**

  DDSTopic *(p. 1419) entity and associated elements*

- **User Data Type Support**

  Defines generic classes and macros to support user data types.

- **Type Code Support**

  <<**eXtension**>> (p. 199) *A* **DDS_TypeCode** (p. 992) *is a mechanism for representing a type at runtime. RTI Connext can use type codes to send type definitions on the network. You will need to understand this API in order to use the* ***Dynamic Data*** *(p. 77) capability or to inspect the type information you receive from remote readers and writers.*

- **Built-in Types**

  <<**eXtension**>> (p. 199) *RTI Connext provides a set of very simple data types for you to use with the topics in your application.*

- **Dynamic Data**

  <<**eXtension**>> (p. 199) *The Dynamic Data API provides a way to interact with arbitrarily complex data types at runtime without the need for code generation.*

- **DDS-Specific Primitive Types**

  Basic DDS value types for use in user data types.

5.6.1 Detailed Description

Contains the **DDSTopic** (p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322) classes, the **DDSTopicListener** (p. 1430) interface, and more generally, all that is needed by an application to define **DDSTopic** (p. 1419) objects and attach QoS policies to them.

5.7 Topics

DDSTopic (p. 1419) entity and associated elements

Classes

- class **DDSTopicDescription**

  <<**interface**>> (p. 199) *Base class for* ***DDSTopic*** *(p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322).*

- class **DDSContentFilteredTopic**

  <<**interface**>> (p. 199) *Specialization of* ***DDSTopicDescription*** *(p. 1427) that allows for content-based subscriptions.*

- class **DDSMultiTopic**

  *[***Not supported (optional)***]* <<**interface**>> (p. 199) *A specialization of* ***DDSTopicDescription*** *(p. 1427) that allows subscriptions that combine/filter/rearrange data coming from several topics.*

- class **DDSTopic**

  <<**interface**>> (p. 199) *The most basic description of the data to be published and subscribed.*

- class **DDSTopicListener**

  <<**interface**>> (p. 199) ***DDSListener*** *(p. 1318) for* ***DDSTopic*** *(p. 1419) entities.*

- class **DDSContentFilter**

  <<**interface**>> (p. 199) *Interface to be used by a custom filter of a* ***DDSContentFilteredTopic*** *(p. 1081)*

- struct **DDS_InconsistentTopicStatus**

  DDS_INCONSISTENT_TOPIC_STATUS *(p. 322)*

- struct **DDS_TopicQos**

  QoS policies supported by a ***DDSTopic*** *(p. 1419) entity.*

5.7.1 Detailed Description

DDSTopic (p. 1419) entity and associated elements

5.8 User Data Type Support

Defines generic classes and macros to support user data types.

Classes

- struct **FooTypeSupport**

  <<**interface**>> (p. 199) <<**generic**>> (p. 199) *User data type specific interface.*

- class **DDSTypeSupport**

  <<**interface**>> (p. 199) *An abstract* marker *interface that has to be specialized for each concrete user data type that will be used by the application.*

- struct **Foo**

  A representative user-defined data type.

- struct **DDS_InstanceHandleSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_InstanceHandle_t*** *(p. 53)* > .

Defines

- #define **DDS_TYPESUPPORT_CPP**(TTypeSupport, TData)

  Declares the interface required to support a user data type.

- #define **DDS_DATAWRITER_CPP_METP**(TDataWriter, TData)

  Declares the interface required to support a user data type specific data writer.

- #define **DDS_DATAREADER_CPP_METP**(TDataReader, TDataSeq, TData)

  Declares the interface required to support a user data type-specific data reader.

Typedefs

- typedef DDS_HANDLE_TYPE_NATIVE **DDS_InstanceHandle_t**

  Type definition for an instance handle.

Functions

- **DDS_Boolean DDS_InstanceHandle_equals** (const **DDS_InstanceHandle_t** *self, const **DDS_InstanceHandle_t** *other)

  Compares this instance handle with another handle for equality.

- **DDS_Boolean DDS_InstanceHandle_is_nil** (const **DDS_InstanceHandle_t** *self)

  *Compare this handle to **DDS_HANDLE_NIL*** (p. 55).

Variables

- const **DDS_InstanceHandle_t DDS_HANDLE_NIL**

  The NIL instance handle.

5.8.1 Detailed Description

Defines generic classes and macros to support user data types.

DDS specifies strongly typed interfaces to read and write user data. For each data class defined by the application, there is a number of specialised classes that are required to facilitate the type-safe interaction of the application with RTI Connext.

RTI Connext provides an automatic means to generate all these type-specific classes with the **rtiddsgen** (p. 220) utility. The complete set of automatic classes created for a hypothetical user data type named `Foo` (p. 1443) are shown below.

The macros defined here declare the strongly typed APIs needed to support an arbitrary user defined data of type `Foo` (p. 1443).

See also:

rtiddsgen (p. 220)

5.8.2 Define Documentation

5.8.2.1 #define DDS_TYPESUPPORT_CPP(TTypeSupport, TData)

Declares the interface required to support a user data type.

Defines:

FooTypeSupport *(p. 1509)* TypeSupport of type `Foo` (p. 1443), i.e. `FooTypeSupport` (p. 1509)

Examples:

HelloWorldSupport.cxx.

5.8.2.2 #define DDS_DATAWRITER_CPP_METP(TDataWriter, TData)

Declares the interface required to support a user data type specific data writer.

Uses:

FooTypeSupport *(p. 1509)* user data type, `Foo` (p. 1443)

Defines:

FooDataWriter *(p. 1475)* **DDSDataWriter** (p. 1113) of type `Foo` (p. 1443), i.e. `FooDataWriter` (p. 1475)

5.8.2.3 #define DDS_DATAREADER_CPP_METP(TDataReader, TDataSeq, TData)

Declares the interface required to support a user data type-specific data reader.

Uses:

FooTypeSupport *(p. 1509)* user data type, `Foo` (p. 1443) **FooSeq** (p. 1494) sequence of user data type, `sequence<::Foo>`

Defines:

FooDataReader *(p. 1444)* **DDSDataReader** (p. 1087) of type `Foo` (p. 1443), i.e. `FooDataReader` (p. 1444)

See also:

FooSeq (p. 1494)

5.8.3 Typedef Documentation

5.8.3.1 typedef DDS_HANDLE_TYPE_NATIVE DDS_InstanceHandle_t

Type definition for an instance handle.

Handle to identiy different instances of the same **DDSTopic** (p. 1419) of a certain type.

See also:

FooDataWriter::register_instance (p. 1478)
DDS_SampleInfo::instance_handle (p. 917)

Examples:

HelloWorld_publisher.cxx.

5.8.4 Function Documentation

5.8.4.1 DDS_Boolean DDS_InstanceHandle_equals (const DDS_InstanceHandle_t * *self*, const DDS_InstanceHandle_t * *other*)

Compares this instance handle with another handle for equality.

Parameters:

self *<<in>> (p. 200)* This handle. Cannot be NULL.
other *<<in>> (p. 200)* The other handle to be compared with this handle. Cannot be NULL.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the two handles have equal values, or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

See also:

DDS_InstanceHandle_is_nil (p. 54)

5.8.4.2 DDS_Boolean DDS_InstanceHandle_is_nil (const DDS_InstanceHandle_t * *self*)

Compare this handle to **DDS_HANDLE_NIL** (p. 55).

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given instance handle is equal to **DDS_HANDLE_NIL** (p. 55) or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

See also:

DDS_InstanceHandle_equals (p. 54)

5.8.5 Variable Documentation

5.8.5.1 const DDS_InstanceHandle_t DDS_HANDLE_NIL

The NIL instance handle.

Special **DDS_InstanceHandle_t** (p. 53) value

See also:

DDS_InstanceHandle_is_nil (p. 54)

Examples:

HelloWorld_publisher.cxx.

5.9 Type Code Support

*<<**eXtension**>> (p. 199)* A ***DDS_TypeCode*** *(p. 992)* is a mechanism for representing a type at runtime. RTI Connext can use type codes to send type definitions on the network. You will need to understand this API in order to use the **Dynamic Data** (p. 77) capability or to inspect the type information you receive from remote readers and writers.

Classes

- struct **DDS_TypeCode**

  *The definition of a particular data type, which you can use to inspect the name, members, and other properties of types generated with **rtiddsgen** (p. 220) or to modify types you define yourself at runtime.*

- struct **DDS_StructMember**

  A description of a member of a struct.

- struct **DDS_StructMemberSeq**

  Defines a sequence of struct members.

- struct **DDS_UnionMember**

  A description of a member of a union.

- struct **DDS_UnionMemberSeq**

  Defines a sequence of union members.

- struct **DDS_EnumMember**

  A description of a member of an enumeration.

- struct **DDS_EnumMemberSeq**

  Defines a sequence of enumerator members.

- struct **DDS_ValueMember**

  A description of a member of a value type.

- struct **DDS_ValueMemberSeq**

  Defines a sequence of value members.

- struct **DDS_TypeCodeFactory**

  A singleton factory for creating, copying, and deleting data type definitions dynamically.

Defines

- #define **DDS_TYPECODE_MEMBER_ID_INVALID**

  *A sentinel indicating an invalid **DDS_TypeCode** (p. 992) member ID.*

- #define **DDS_TYPECODE_INDEX_INVALID**

  *A sentinel indicating an invalid **DDS_TypeCode** (p. 992) member index.*

- #define **DDS_TYPECODE_NOT_BITFIELD**

  Indicates that a member of a type is not a bitfield.

- #define **DDS_VM_NONE**

  Constant used to indicate that a value type has no modifiers.

- #define **DDS_VM_CUSTOM**

  Constant used to indicate that a value type has the `custom` *modifier.*

- #define **DDS_VM_ABSTRACT**

  Constant used to indicate that a value type has the `abstract` *modifier.*

- #define **DDS_VM_TRUNCATABLE**

  Constant used to indicate that a value type has the `truncatable` *modifier.*

- #define **DDS_PRIVATE_MEMBER**

  Constant used to indicate that a value type member is private.

- #define **DDS_PUBLIC_MEMBER**

  Constant used to indicate that a value type member is public.

- #define **DDS_TYPECODE_NONKEY_MEMBER**

  A flag indicating that a type member is optional and not part of the key.

- #define **DDS_TYPECODE_KEY_MEMBER**

  A flag indicating that a type member is part of the key for that type, and therefore required.

- #define **DDS_TYPECODE_NONKEY_REQUIRED_MEMBER**

  A flag indicating that a type member is not part of the key but is nevertheless required.

Typedefs

- typedef short **DDS_ValueModifier**

  Modifier type for a value type.

- typedef short **DDS_Visibility**

  Type to indicate the visibility of a value type member.

Enumerations

- enum **DDS_TCKind** {
  DDS_TK_NULL,
  DDS_TK_SHORT,
  DDS_TK_LONG,
  DDS_TK_USHORT,
  DDS_TK_ULONG,
  DDS_TK_FLOAT,
  DDS_TK_DOUBLE,
  DDS_TK_BOOLEAN,
  DDS_TK_CHAR,
  DDS_TK_OCTET,
  DDS_TK_STRUCT,
  DDS_TK_UNION,
  DDS_TK_ENUM,
  DDS_TK_STRING,
  DDS_TK_SEQUENCE,
  DDS_TK_ARRAY,
  DDS_TK_ALIAS,
  DDS_TK_LONGLONG,
  DDS_TK_ULONGLONG,
  DDS_TK_LONGDOUBLE,
  DDS_TK_WCHAR,
  DDS_TK_WSTRING,
  DDS_TK_VALUE,
  DDS_TK_SPARSE }

  *Enumeration type for **DDS_TypeCode** (p. 992) kinds.*

- enum **DDS_ExtensibilityKind** {
  DDS_FINAL_EXTENSIBILITY,
  DDS_EXTENSIBLE_EXTENSIBILITY,
  DDS_MUTABLE_EXTENSIBILITY }
  Type to indicate the extensibility of a type.

Variables

- **DDS_TypeCode DDS_g_tc_null**
  Basic NULL type.
- **DDS_TypeCode DDS_g_tc_long**
  Basic 32-bit signed integer type.
- **DDS_TypeCode DDS_g_tc_ushort**
  Basic unsigned 16-bit integer type.
- **DDS_TypeCode DDS_g_tc_ulong**
  Basic unsigned 32-bit integer type.
- **DDS_TypeCode DDS_g_tc_float**
  Basic 32-bit floating point type.
- **DDS_TypeCode DDS_g_tc_double**
  Basic 64-bit floating point type.
- **DDS_TypeCode DDS_g_tc_boolean**
  Basic Boolean type.
- **DDS_TypeCode DDS_g_tc_octet**
  Basic octet/byte type.
- **DDS_TypeCode DDS_g_tc_longlong**
  Basic 64-bit integer type.
- **DDS_TypeCode DDS_g_tc_ulonglong**
  Basic unsigned 64-bit integer type.
- **DDS_TypeCode DDS_g_tc_longdouble**

Basic 128-bit floating point type.

- **DDS_TypeCode DDS_g_tc_wchar**

  Basic four-byte character type.

5.9.1 Detailed Description

*<<**eXtension**>> (p. 199)* A ***DDS_TypeCode*** *(p. 992)* is a mechanism for representing a type at runtime. RTI Connext can use type codes to send type definitions on the network. You will need to understand this API in order to use the **Dynamic Data** (p. 77) capability or to inspect the type information you receive from remote readers and writers.

Type codes are values that are used to describe arbitrarily complex types at runtime. Type code values are manipulated via the **DDS_TypeCode** (p. 992) class, which has an analogue in CORBA.

A **DDS_TypeCode** (p. 992) value consists of a type code *kind* (represented by the **DDS_TCKind** (p. 66) enumeration) and a list of *members* (that is, fields). These members are recursive: each one has its own **DDS_TypeCode** (p. 992), and in the case of complex types (structures, arrays, and so on), these contained type codes contain their own members.

There are a number of uses for type codes. The type code mechanism can be used to unambiguously match type representations. The **DDS_TypeCode::equal** (p. 998) method is a more reliable test than comparing the string type names, requiring equivalent definitions of the types.

5.9.2 Accessing a Local ::DDS_TypeCode

When generating types with **rtiddsgen** (p. 220), type codes are enabled by default. (The *-notypecode* option can be used to disable generation of **DDS_TypeCode** (p. 992) information.) For these types, a **DDS_TypeCode** (p. 992) may be accessed by calling the `Foo_get_typecode` function for a type "Foo", which returns a **DDS_TypeCode** (p. 992) pointer.

This API also includes support for dynamic creation of **DDS_TypeCode** (p. 992) values, typically for use with the **Dynamic Data** (p. 77) API. You can create a **DDS_TypeCode** (p. 992) using the **DDS_TypeCodeFactory** (p. 1022) class. You will construct the **DDS_TypeCode** (p. 992) recursively, from the outside in: start with the type codes for primitive types, then compose them into complex types like arrays, structures, and so on. You will find the following methods helpful:

- **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026), which provides

the **DDS_TypeCode** (p. 992) instances corresponding to the primitive types (e.g. **DDS_TK_LONG** (p. 66), **DDS_TK_SHORT** (p. 66), and so on).

- **DDS_TypeCodeFactory::create_string_tc** (p. 1033) and **DDS_TypeCodeFactory::create_wstring_tc** (p. 1034) create a **DDS_TypeCode** (p. 992) representing a text string with a certain *bound* (i.e. maximum length).

- **DDS_TypeCodeFactory::create_array_tc** (p. 1035) and **DDS_TypeCodeFactory::create_sequence_tc** (p. 1034) create a **DDS_TypeCode** (p. 992) for a collection based on the **DDS_TypeCode** (p. 992) for its elements.

- **DDS_TypeCodeFactory::create_struct_tc** (p. 1027), **DDS_TypeCodeFactory::create_value_tc** (p. 1028), and **DDS_TypeCodeFactory::create_sparse_tc** (p. 1036) create a **DDS_TypeCode** (p. 992) for a structured type.

5.9.3 Accessing a Remote ::DDS_TypeCode

In addition to being used locally, RTI Connext can transmit **DDS_TypeCode** (p. 992) on the network between participants. This information can be used to access information about types used remotely at runtime, for example to be able to publish or subscribe to topics of arbitrarily types (see **Dynamic Data** (p. 77)). This functionality is useful for a generic system monitoring tool like `rtiddsspy`.

Remote **DDS_TypeCode** (p. 992) information is shared during discovery over the publication and subscription built-in topics and can be accessed using the built-in readers for these topics; see **Built-in Topics** (p. 42). Discovered **DDS_TypeCode** (p. 992) values are not cached by RTI Connext upon receipt and are therefore not available from the built-in topic data returned by **DDSDataWriter::get_matched_subscription_data** (p. 1124) or **DDSDataReader::get_matched_publication_data** (p. 1097).

The space available locally to deserialize a discovered remote **DDS_TypeCode** (p. 992) is specified by the **DDSDomainParticipant** (p. 1139)'s **DDS_DomainParticipantResourceLimitsQosPolicy::type_code_max_serialized_length** (p. 608) QoS parameter. To support especially complex type codes, it may be necessary for you to increase the value of this parameter.

See also:

DDS_TypeCode (p. 992)
Dynamic Data (p. 77)
rtiddsgen (p. 220)

DDS_SubscriptionBuiltinTopicData (p. 936)
DDS_PublicationBuiltinTopicData (p. 839)

5.9.4 Define Documentation

5.9.4.1 #define DDS_TYPECODE_MEMBER_ID_INVALID

A sentinel indicating an invalid **DDS_TypeCode** (p. 992) member ID.

5.9.4.2 #define DDS_TYPECODE_INDEX_INVALID

A sentinel indicating an invalid **DDS_TypeCode** (p. 992) member index.

5.9.4.3 #define DDS_TYPECODE_NOT_BITFIELD

Indicates that a member of a type is not a bitfield.

5.9.4.4 #define DDS_VM_NONE

Constant used to indicate that a value type has no modifiers.

See also:

DDS_ValueModifier (p. 65)

Examples:

HelloWorld.cxx.

5.9.4.5 #define DDS_VM_CUSTOM

Constant used to indicate that a value type has the `custom` modifier.

This modifier is used to specify whether the value type uses custom marshaling.

See also:

DDS_ValueModifier (p. 65)

5.9.4.6 #define DDS_VM_ABSTRACT

Constant used to indicate that a value type has the `abstract` modifier.

An abstract value type may not be instantiated.

See also:

DDS_ValueModifier (p. 65)

5.9.4.7 #define DDS_VM_TRUNCATABLE

Constant used to indicate that a value type has the `truncatable` modifier.

A value with a state that derives from another value with a state can be declared as truncatable. A truncatable type means the object can be truncated to the base type.

See also:

DDS_ValueModifier (p. 65)

5.9.4.8 #define DDS_PRIVATE_MEMBER

Constant used to indicate that a value type member is private.

See also:

DDS_Visibility (p. 66)
DDS_PUBLIC_MEMBER (p. 63)

Examples:

HelloWorld.cxx.

5.9.4.9 #define DDS_PUBLIC_MEMBER

Constant used to indicate that a value type member is public.

See also:

DDS_Visibility (p. 66)
DDS_PRIVATE_MEMBER (p. 63)

5.9.4.10 #define DDS_TYPECODE_NONKEY_MEMBER

A flag indicating that a type member is optional and not part of the key.

Only sparse value types (i.e. types of **DDS_TCKind** (p. 66) **DDS_TK_SPARSE** (p. 67)) support this flag. Non-key members of other type kinds should use the flag **DDS_TYPECODE_NONKEY_REQUIRED_MEMBER** (p. 65).

If a type is used with the **Dynamic Data** (p. 77) facility, a **DDS_DynamicData** (p. 622) sample of the type will only contain a value for a **DDS_TYPECODE_NONKEY_MEMBER** (p. 64) field if one has been explicitly set (see, for example, **DDS_DynamicData::set_long** (p. 687)). The middleware will *not* assume any default value.

See also:

DDS_TYPECODE_KEY_MEMBER (p. 64)
DDS_TYPECODE_NONKEY_REQUIRED_MEMBER (p. 65)
DDS_TYPECODE_KEY_MEMBER (p. 64)
DDS_TypeCode::add_member (p. 1017)
DDS_TypeCode::add_member_ex (p. 1019)
DDS_TypeCode::is_member_key (p. 1005)
DDS_TypeCode::is_member_required (p. 1005)
DDS_StructMember::is_key (p. 932)
DDS_ValueMember::is_key (p. 1051)

5.9.4.11 #define DDS_TYPECODE_KEY_MEMBER

A flag indicating that a type member is part of the key for that type, and therefore required.

If a type is used with the **Dynamic Data** (p. 77) facility, all **DDS_DynamicData** (p. 622) samples of the type will contain a value for all **DDS_TYPECODE_KEY_MEMBER** (p. 64) fields, even if the type is a sparse value type (i.e. of kind **DDS_TK_SPARSE** (p. 67)). If you do not set a value of the member explicitly (see, for example, **DDS_DynamicData::set_long** (p. 687)), the middleware will assume a default "zero" value: numeric values will be set to zero; strings and sequences will be of zero length.

See also:

DDS_TYPECODE_NONKEY_REQUIRED_MEMBER (p. 65)
DDS_TYPECODE_NONKEY_MEMBER (p. 64)
DDS_TypeCode::add_member (p. 1017)
DDS_TypeCode::add_member_ex (p. 1019)
DDS_TypeCode::is_member_key (p. 1005)

DDS_TypeCode::is_member_required (p. 1005)
DDS_StructMember::is_key (p. 932)
DDS_ValueMember::is_key (p. 1051)

5.9.4.12 #define DDS_TYPECODE_NONKEY_REQUIRED_MEMBER

A flag indicating that a type member is not part of the key but is nevertheless required.

This is the most common kind of member.

If a type is used with the **Dynamic Data** (p. 77) facility, all **DDS_DynamicData** (p. 622) samples of the type will contain a value for all **DDS_TYPECODE_NONKEY_REQUIRED_MEMBER** (p. 65) fields, even if the type is a sparse value type (i.e. of kind **DDS_TK_SPARSE** (p. 67)). If you do not set a value of the member explicitly (see, for example, **DDS_DynamicData::set_long** (p. 687)), the middleware will assume a default "zero" value: numeric values will be set to zero; strings and sequences will be of zero length.

See also:

DDS_TYPECODE_KEY_MEMBER (p. 64)
DDS_TYPECODE_NONKEY_MEMBER (p. 64)
DDS_TYPECODE_KEY_MEMBER (p. 64)
DDS_TypeCode::add_member (p. 1017)
DDS_TypeCode::add_member_ex (p. 1019)
DDS_TypeCode::is_member_key (p. 1005)
DDS_TypeCode::is_member_required (p. 1005)
DDS_StructMember::is_key (p. 932)
DDS_ValueMember::is_key (p. 1051)

5.9.5 Typedef Documentation

5.9.5.1 typedef short DDS_ValueModifier

Modifier type for a value type.

See also:

DDS_VM_NONE (p. 62)
DDS_VM_CUSTOM (p. 62)
DDS_VM_ABSTRACT (p. 63)
DDS_VM_TRUNCATABLE (p. 63)

5.9.5.2 typedef short DDS_Visibility

Type to indicate the visibility of a value type member.

See also:

DDS_PRIVATE_MEMBER (p. 63)
DDS_PUBLIC_MEMBER (p. 63)

5.9.6 Enumeration Type Documentation

5.9.6.1 enum DDS_TCKind

Enumeration type for **DDS_TypeCode** (p. 992) kinds.

Type code kinds are modeled as values of this type.

Enumerator:

DDS_TK_NULL Indicates that a type code does not describe anything.
DDS_TK_SHORT short type.
DDS_TK_LONG long type.
DDS_TK_USHORT unsigned short type.
DDS_TK_ULONG unsigned long type.
DDS_TK_FLOAT float type.
DDS_TK_DOUBLE double type.
DDS_TK_BOOLEAN boolean type.
DDS_TK_CHAR char type.
DDS_TK_OCTET octet type.
DDS_TK_STRUCT struct type.
DDS_TK_UNION union type.
DDS_TK_ENUM enumerated type.
DDS_TK_STRING string type.
DDS_TK_SEQUENCE sequence type.
DDS_TK_ARRAY array type.
DDS_TK_ALIAS alias (typedef) type.
DDS_TK_LONGLONG long long type.
DDS_TK_ULONGLONG unsigned long long type.
DDS_TK_LONGDOUBLE long double type.
DDS_TK_WCHAR wide char type.

DDS_TK_WSTRING wide string type.

DDS_TK_VALUE value type.

DDS_TK_SPARSE A sparse value type.

A sparse value type is one in which all of the fields are not necessarily sent on the network as a part of every sample.

Fields of a sparse value type fall into one of three categories:

- Key fields (see **DDS_TYPECODE_KEY_MEMBER** (p. 64))
- Non-key, but required members (see **DDS_TYPECODE_NONKEY_REQUIRED_MEMBER** (p. 65))
- Non-key, optional members (see **DDS_TYPECODE_NONKEY_MEMBER** (p. 64))

Fields of the first two kinds must appear in every sample. These are also the only kinds of fields on which you can perform content filtering (see **DDSContentFilteredTopic** (p. 1081)), because filter evaluation on a non-existent field is not well defined.

5.9.6.2 enum DDS_ExtensibilityKind

Type to indicate the extensibility of a type.

Enumerator:

DDS_FINAL_EXTENSIBILITY Specifies that a type has FINAL extensibility.

A type may be final, indicating that the range of its possible values is strictly defined. In particular, it is not possible to add elements to members of collection or aggregated types while maintaining type assignability.

The following types are always final:

- **DDS_TK_ARRAY** (p. 66)
- All primitive types

DDS_EXTENSIBLE_EXTENSIBILITY Specifies that a type has EXTENSIBLE extensibility.

A type may be extensible, indicating that two types, where one contains all of the elements/members of the other plus additional elements/members appended to the end, may remain assignable.

DDS_MUTABLE_EXTENSIBILITY Specifies that a type has MUTABLE extensibility.

[**Not supported.**]

A type may be mutable, indicating that two types may differ from one another in the additional, removal, and/or transposition of elements/members while remaining assignable.

The following types are always mutable:

- **DDS_TK_SEQUENCE** (p. 66)
- **DDS_TK_STRING** (p. 66)
- **DDS_TK_WSTRING** (p. 67)

The support for type mutability in this release is limited to **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRING** (p. 66), and **DDS_TK_WSTRING** (p. 67).

Mutability in **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_VALUE** (p. 67), and **DDS_TK_ENUM** (p. 66) is not supported.

5.9.7 Variable Documentation

5.9.7.1 DDS_TypeCode DDS_g_tc_null

Basic NULL type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)

5.9.7.2 DDS_TypeCode DDS_g_tc_long

Basic 32-bit signed integer type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_Long (p. 300)

5.9.7.3 DDS_TypeCode DDS_g_tc_ushort

Basic unsigned 16-bit integer type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_UnsignedShort (p. 299)

5.9.7.4 DDS_TypeCode DDS_g_tc_ulong

Basic unsigned 32-bit integer type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_UnsignedLong (p. 300)

5.9.7.5 DDS_TypeCode DDS_g_tc_float

Basic 32-bit floating point type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_Float (p. 300)

5.9.7.6 DDS_TypeCode DDS_g_tc_double

Basic 64-bit floating point type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_Double (p. 300)

5.9.7.7 DDS_TypeCode DDS_g_tc_boolean

Basic Boolean type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_Boolean (p. 301)

5.9.7.8 DDS_TypeCode DDS_g_tc_octet

Basic octet/byte type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_Octet (p. 299)

5.9.7.9 DDS_TypeCode DDS_g_tc_longlong

Basic 64-bit integer type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_LongLong (p. 300)

5.9.7.10 DDS_TypeCode DDS_g_tc_ulonglong

Basic unsigned 64-bit integer type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_UnsignedLongLong (p. 300)

5.9.7.11 DDS_TypeCode DDS_g_tc_longdouble

Basic 128-bit floating point type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_LongDouble (p. 300)

5.9.7.12 DDS_TypeCode DDS_g_tc_wchar

Basic four-byte character type.

For new code, **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026) is preferred to using this global variable.

See also:

DDS_TypeCodeFactory::get_primitive_tc (p. 1026)
DDS_Wchar (p. 299)

5.10 Built-in Types

<<*eXtension*>> *(p. 199)* RTI Connext provides a set of very simple data types for you to use with the topics in your application.

Modules

- **String Built-in Type**

  Built-in type consisting of a single character string.

- **KeyedString Built-in Type**

  Built-in type consisting of a string payload and a second string that is the key.

- **Octets Built-in Type**

  Built-in type consisting of a variable-length array of opaque bytes.

- **KeyedOctets Built-in Type**

  Built-in type consisting of a variable-length array of opaque bytes and a string that is the key.

5.10.1 Detailed Description

<<*eXtension*>> *(p. 199)* RTI Connext provides a set of very simple data types for you to use with the topics in your application.

The middleware provides four built-in types:

- String: A payload consisting of a single string of characters. This type has no key.
- **DDS_KeyedString** (p. 768): A payload consisting of a single string of characters and a second string, the key, that identifies the instance to which the sample belongs.
- **DDS_Octets** (p. 799): A payload consisting of an opaque variable-length array of bytes. This type has no key.
- **DDS_KeyedOctets** (p. 765): A payload consisting of an opaque variable-length array of bytes and a string, the key, that identifies the instance to which the sample belongs.

The String and **DDS_KeyedString** (p. 768) types are appropriate for simple text-based applications. The **DDS_Octets** (p. 799) and **DDS_KeyedOctets** (p. 765) types are appropriate for applications that perform their own custom data serialization, such as legacy applications still in the process of migrating to RTI Connext. In most cases, string-based or structured data is preferable to opaque data, because the latter cannot be easily visualized in tools or used with content-based filters (see **DDSContentFilteredTopic** (p. 1081)).

The built-in types are very simple in order to get you up and running as quickly as possible. If you need a structured data type you can define your own type with exactly the fields you need in one of two ways:

- At compile time, by generating code from an IDL or XML file using the **rtiddsgen** (p. 220) utility
- At runtime, by using the **Dynamic Data** (p. 77) API

5.10.2 Managing Memory for Builtin Types

When a sample is written, the DataWriter serializes it and stores the result in a buffer obtained from a pool of preallocated buffers. In the same way, when a sample is received, the DataReader deserializes it and stores the result in a sample coming from a pool of preallocated samples.

For builtin types, the maximum size of the buffers/samples and depends on the nature of the application using the builtin type.

You can configure the maximum size of the builtin types on a per-DataWriter and per-DataReader basis using the **DDS_PropertyQosPolicy** (p. 834) in DataWriters, DataReaders or Participants.

The following table lists the supported builtin type properties to configure memory allocation. When the properties are defined in the DomainParticipant, they are applicable to all DataWriters and DataReaders belonging to the DomainParticipant unless they are overwrittem in the DataWriters and DataReaders.

The previous properties must be set consistently with respect to the corresponding $*$.max_size properties that set the maximum size of the builtin types in the typecode.

5.10.3 Typecodes for Builtin Types

The typecodes associated with the builtin types are generated from the following IDL type definitions:

```
module DDS {
     struct String {
         string value;
     };

     struct KeyedString {
         string key;
         string value;
     };

     struct Octets {
         sequence<octet> value;
     };

     struct KeyedOctets {
         string key;
         sequence<octet> value;
     };
 };
```

The maximum size of the strings and sequences that will be included in the type code definitions can be configured on a per-DomainParticipant-basis by using the properties in following table.

For more information about the built-in types, including how to control memory usage and maximum lengths, please see chapter 3, *Data Types and Data Samples*, in the `User's Manual`.

Property	Description
dds.builtin_type.string.alloc_size	Maximum size of the strings published by the **DDSStringDataWriter** (p. 1383) or received the **DDSStringDataReader** (p. 1379) (includes the NULL-terminated character). Default: dds.builtin_type.string.max_size if defined. Otherwise, 1024.
dds.builtin_type.keyed_string.alloc_key_size	Maximum size of the keys used by the **DDSKeyedStringDataWriter** (p. 1304) or **DDSKeyedStringDataReader** (p. 1293) (includes the NULL-terminated character). Default: dds.builtin_type.keyed_string.max_key_size if defined. Otherwise, 1024.
dds.builtin_type.keyed_string.alloc_size	Maximum size of the strings published by the **DDSKeyedStringDataWriter** (p. 1304) or received by the **DDSKeyedStringDataReader** (p. 1293) (includes the NULL-terminated character). Default: dds.builtin_type.keyed_string.max_size if defined. Otherwise, 1024.
dds.builtin_type.octets.alloc_size	Maximum size of the octet sequences published the **DDSOctetsDataWriter** (p. 1331) or received by the **DDSOctetsDataReader** (p. 1326). Default: dds.builtin_type.octets.max_size if defined. Otherwise, 2048.
dds.builtin_type.keyed_octets.alloc_key_size	Maximum size of the key published by the **DDSKeyedOctetsDataWriter** (p. 1276) or received by the **DDSKeyedOctetsDataReader** (p. 1265) (includes the NULL-terminated character). Default: dds.builtin_type.keyed_octets.max_key_size if defined. Otherwise, 1024.
dds.builtin_type.keyed_octets.alloc_size	Maximum size of the octets sequences published by a **DDSKeyedOctetsDataWriter** (p. 1276) or received by a **DDSKeyedOctetsDataReader** (p. 1265).

Property	Description
dds.builtin_type.string.max_size	Maximum size of the strings published by the StringDataWriters and received by the StringDataReaders belonging to a DomainParticipant (includes the NULL-terminated character). Default: 1024.
dds.builtin_type.keyed_string.max_key_size	Maximum size of the keys used by the KeyedStringDataWriters and KeyedStringDataReaders belonging to a DomainParticipant (includes the NULL-terminated character). Default: 1024.
dds.builtin_type.keyed_string.max_size	Maximum size of the strings published by the KeyedStringDataWriters and received by the KeyedStringDataReaders belonging to a DomainParticipant using the builtin type (includes the NULL-terminated character). Default: 1024
dds.builtin_type.octets.max_size	Maximum size of the octet sequences published by the OctetsDataWriters and received by the OctetsDataReader belonging to a DomainParticipant. Default: 2048
dds.builtin_type.keyed_octets.max_key_size	Maximum size of the keys used by the KeyedOctetsStringDataWriters and KeyedOctetsStringDataReaders belonging to a DomainParticipant (includes the NULL-terminated character). Default: 1024.
dds.builtin_type.keyed_octets.max_size	Maximum size of the octet sequences published by the KeyedOctetsDataWriters and received by the KeyedOctetsDataReaders belonging to a DomainParticipant. Default: 2048

Table 5.4: *Properties for Allocating Size of Builtin Types, per DomainParticipant*

5.11 Dynamic Data

<<***eXtension***>> *(p. 199)* The Dynamic Data API provides a way to interact with arbitrarily complex data types at runtime without the need for code generation.

Classes

- class **DDSDynamicDataTypeSupport**

  A factory for registering a dynamically defined type and creating ***DDS_DynamicData*** *(p. 622) objects.*

- class **DDSDynamicDataReader**

  Reads (subscribes to) objects of type ***DDS_DynamicData*** *(p. 622).*

- class **DDSDynamicDataWriter**

  Writes (publishes) objects of type ***DDS_DynamicData*** *(p. 622).*

- struct **DDS_DynamicDataProperty_t**

  A collection of attributes used to configure ***DDS_DynamicData*** *(p. 622) objects.*

- struct **DDS_DynamicDataTypeSerializationProperty_t**

  Properties that govern how data of a certain type will be serialized on the network.

- struct **DDS_DynamicDataInfo**

  A descriptor for a ***DDS_DynamicData*** *(p. 622) object.*

- struct **DDS_DynamicDataMemberInfo**

  A descriptor for a single member (i.e. field) of dynamically defined data type.

- struct **DDS_DynamicData**

  A sample of any complex data type, which can be inspected and manipulated reflectively.

- struct **DDS_DynamicDataSeq**

  An ordered collection of ***DDS_DynamicData*** *(p. 622) elements.*

- struct **DDS_DynamicDataTypeProperty_t**

  A collection of attributes used to configure ***DDSDynamicDataTypeSupport*** *(p. 1246) objects.*

Defines

- #define **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED**

  A sentinel value that indicates that no member ID is needed in order to perform some operation.

Typedefs

- typedef **DDS_Long DDS_DynamicDataMemberId**

  An integer that uniquely identifies some member of a data type within that type.

Variables

- struct **DDS_DynamicDataProperty_t DDS_DYNAMIC_DATA_PROPERTY_DEFAULT**

  *Sentinel constant indicating default values for **DDS_DynamicDataProperty_t** (p. 725).*

- struct **DDS_DynamicDataTypeProperty_t DDS_DYNAMIC_DATA_TYPE_PROPERTY_DEFAULT**

  *Sentinel constant indicating default values for **DDS_DynamicDataTypeProperty_t** (p. 728).*

5.11.1 Detailed Description

<<***eXtension***>> *(p. 199)* The Dynamic Data API provides a way to interact with arbitrarily complex data types at runtime without the need for code generation.

This API allows you to define new data types, modify existing data types, and interact reflectively with samples. To use it, you will take the following steps:

1. Obtain a DDS_TypeCode (p. 992) (see Type Code Support (p. 56)) that defines the type definition you want to use.

A **DDS_TypeCode** (p. 992) includes a type's *kind* (**DDS_TCKind** (p. 66)), *name*, and *members* (that is, fields). You can create your own **DDS_TypeCode** (p. 992) using the **DDS_TypeCodeFactory** (p. 1022) class – see, for example, the **DDS_TypeCodeFactory::create_struct_tc** (p. 1027) method. Alternatively, you can use a remote **DDS_TypeCode** (p. 992) that you discovered

on the network (see **Built-in Topics** (p. 42)) or one generated by **rtiddsgen** (p. 220).

2. Wrap the DDS_TypeCode (p. 992) in a DDSDynamicDataTypeSupport (p. 1246) object.

See the constructor **DDSDynamicDataTypeSupport::DDSDynamicDataTypeSupport** (p. 1247). This object lets you connect the type definition to a **DDSDomainParticipant** (p. 1139) and manage data samples (of type **DDS_DynamicData** (p. 622)).

3. Register the DDSDynamicDataTypeSupport (p. 1246) with one or more domain participants.

See **DDSDynamicDataTypeSupport::register_type** (p. 1249). This action associates the data type with a logical name that you can use to create topics. (Starting with this step, working with a dynamically defined data type is almost exactly the same as working with a generated one.)

4. Create a DDSTopic (p. 1419) from the DDSDomainParticipant (p. 1139).

Use the name under which you registered your data type – see **DDSDomainParticipant::create_topic** (p. 1175). This **DDSTopic** (p. 1419) is what you will use to produce and consume data.

5. Create a DDSDynamicDataWriter (p. 1252) and/or DDSDynamicDataReader (p. 1245).

These objects will produce and/or consume data (of type **DDS_DynamicData** (p. 622)) on the **DDSTopic** (p. 1419). You can create these objects directly from the **DDSDomainParticipant** (p. 1139) – see **DDSDomainParticipant::create_datawriter** (p. 1204) and **DDSDomainParticipant::create_datareader** (p. 1208) – or by first creating intermediate **DDSPublisher** (p. 1346) and **DDSSubscriber** (p. 1390) objects – see **DDSDomainParticipant::create_publisher** (p. 1169) and **DDSDomainParticipant::create_subscriber** (p. 1172).

6. Write and/or read the data of interest.

7. Tear down the objects described above.

You should delete them in the reverse order in which you created them. Note that unregistering your data type with the **DDSDomainParticipant** (p. 1139) is optional; all types are automatically unregistered when the **DDSDomainParticipant** (p. 1139) itself is deleted.

5.11.2 Define Documentation

5.11.2.1 #define DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED

A sentinel value that indicates that no member ID is needed in order to perform some operation.

Most commonly, this constant will be used in "get" operations to indicate that a lookup should be performed based on a name, not on an ID.

See also:

DDS_DynamicDataMemberId (p. 80)

5.11.3 Typedef Documentation

5.11.3.1 typedef DDS_Long DDS_DynamicDataMemberId

An integer that uniquely identifies some member of a data type within that type.

The range of a member ID is the range of an unsigned short integer, except for the value 0, which is reserved.

See also:

DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED (p. 80)

5.11.4 Variable Documentation

5.11.4.1 struct DDS_DynamicDataProperty_t DDS_DYNAMIC_DATA_PROPERTY_DEFAULT

Sentinel constant indicating default values for **DDS_DynamicDataProperty_t** (p. 725).

Pass this object instead of your own **DDS_DynamicDataProperty_t** (p. 725) object to use the default property values:

```
DDS_DynamicData* sample = new DDS_DynamicData(
    myTypeCode,
    &DDS_DYNAMIC_DATA_PROPERTY_DEFAULT);
```

See also:

DDS_DynamicDataProperty_t (p. 725)

5.11.4.2 struct DDS_DynamicDataTypeProperty_t DDS_DYNAMIC_DATA_TYPE_PROPERTY_DEFAULT

Sentinel constant indicating default values for **DDS_DynamicDataTypeProperty_t** (p. 728).

Pass this object instead of your own **DDS_DynamicDataTypeProperty_t** (p. 728) object to use the default property values:

```
DDS_DynamicDataTypeSupport* support = new DDS_DynamicDataTypeSupport(
     myTypeCode,
     &DDS_DYNAMIC_DATA_TYPE_PROPERTY_DEFAULT);
```

See also:

DDS_DynamicDataTypeProperty_t (p. 728)

5.12 Publication Module

Contains the **DDSFlowController** (p. 1259), **DDSPublisher** (p. 1346), and **DDSDataWriter** (p. 1113) classes as well as the **DDSPublisherListener** (p. 1370) and **DDSDataWriterListener** (p. 1133) interfaces, and more generally, all that is needed on the publication side.

Modules

- **Publishers**

  DDSPublisher *(p. 1346) entity and associated elements*

- **Data Writers**

  DDSDataWriter *(p. 1113) entity and associated elements*

- **Flow Controllers**

  <<**eXtension**>> (p. 199) ***DDSFlowController*** *(p. 1259) and associated elements*

5.12.1 Detailed Description

Contains the **DDSFlowController** (p. 1259), **DDSPublisher** (p. 1346), and **DDSDataWriter** (p. 1113) classes as well as the **DDSPublisherListener** (p. 1370) and **DDSDataWriterListener** (p. 1133) interfaces, and more generally, all that is needed on the publication side.

5.13 Publishers

DDSPublisher (p. 1346) entity and associated elements

Classes

- class **DDSPublisherListener**

  <<**interface**>> (p. 199) ***DDSListener*** *(p. 1318) for* ***DDSPublisher*** *(p. 1346) status.*

- class **DDSPublisherSeq**

  Declares IDL `sequence` *<* ***DDSPublisher*** *(p. 1346) > .*

- class **DDSPublisher**

  <<**interface**>> (p. 199) *A publisher is the object responsible for the actual dissemination of publications.*

- struct **DDS_PublisherQos**

  QoS policies supported by a ***DDSPublisher*** *(p. 1346) entity.*

Variables

- struct **DDS_DataWriterQos DDS_DATAWRITER_QOS_DEFAULT**

  Special value for creating ***DDSDataWriter*** *(p. 1113) with default QoS.*

- struct **DDS_DataWriterQos DDS_DATAWRITER_QOS_USE_TOPIC_QOS**

  Special value for creating ***DDSDataWriter*** *(p. 1113) with a combination of the default* ***DDS_DataWriterQos*** *(p. 553) and the* ***DDS_TopicQos*** *(p. 965).*

5.13.1 Detailed Description

DDSPublisher (p. 1346) entity and associated elements

5.13.2 Variable Documentation

5.13.2.1 struct DDS_DataWriterQos DDS_DATAWRITER_QOS_DEFAULT

Special value for creating **DDSDataWriter** (p. 1113) with default QoS.

When used in **DDSPublisher::create_datawriter** (p. 1355), this special value is used to indicate that the **DDSDataWriter** (p. 1113) should be created with the default **DDSDataWriter** (p. 1113) QoS by means of the operation get_default_datawriter_qos and using the resulting QoS to create the **DDSDataWriter** (p. 1113).

When used in **DDSPublisher::set_default_datawriter_qos** (p. 1351), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSPublisher::set_default_datawriter_qos** (p. 1351) operation had never been called.

When used in **DDSDataWriter::set_qos** (p. 1126), this special value is used to indicate that the QoS of the **DDSDataWriter** (p. 1113) should be changed to match the current defualt QoS set in the **DDSPublisher** (p. 1346) that the **DDSDataWriter** (p. 1113) belongs to.

Note: You cannot use this value to *get* the default QoS values for a DataWriter; for this purpose, use **DDSDomainParticipant::get_default_datawriter_qos** (p. 1150).

See also:

DDSPublisher::create_datawriter (p. 1355)
DDSPublisher::set_default_datawriter_qos (p. 1351)
DDSDataWriter::set_qos (p. 1126)

Examples:

HelloWorld_publisher.cxx.

5.13.2.2 struct DDS_DataWriterQos DDS_DATAWRITER_QOS_USE_TOPIC_QOS

Special value for creating **DDSDataWriter** (p. 1113) with a combination of the default **DDS_DataWriterQos** (p. 553) and the **DDS_TopicQos** (p. 965).

The use of this value is equivalent to the application obtaining the default **DDS_DataWriterQos** (p. 553) and the **DDS_TopicQos** (p. 965) (by means of the operation **DDSTopic::get_qos** (p. 1423)) and then combining these two QoS using the operation **DDSPublisher::copy_from_topic_qos** (p. 1364) whereby any policy that is set on the **DDS_TopicQos** (p. 965) "overrides" the

corresponding policy on the default QoS. The resulting QoS is then applied to the creation of the **DDSDataWriter** (p. 1113).

This value should only be used in **DDSPublisher::create_datawriter** (p. 1355).

See also:

DDSPublisher::create_datawriter (p. 1355)
DDSPublisher::get_default_datawriter_qos (p. 1350)
DDSTopic::get_qos (p. 1423)
DDSPublisher::copy_from_topic_qos (p. 1364)

5.14 Data Writers

DDSDataWriter (p. 1113) entity and associated elements

Classes

- struct **FooDataWriter**

  <<**interface**>> (p. 199) <<**generic**>> (p. 199) *User data type specific data writer.*

- class **DDSDataWriterListener**

  <<**interface**>> (p. 199) ***DDSListener*** *(p. 1318) for writer status.*

- class **DDSDataWriter**

  <<**interface**>> (p. 199) *Allows an application to set the value of the data to be published under a given* ***DDSTopic*** *(p. 1419).*

- struct **DDS_OfferedDeadlineMissedStatus**

  DDS_OFFERED_DEADLINE_MISSED_STATUS *(p. 323)*

- struct **DDS_LivelinessLostStatus**

  DDS_LIVELINESS_LOST_STATUS *(p. 325)*

- struct **DDS_OfferedIncompatibleQosStatus**

  DDS_OFFERED_INCOMPATIBLE_QOS_STATUS *(p. 323)*

- struct **DDS_PublicationMatchedStatus**

  DDS_PUBLICATION_MATCHED_STATUS *(p. 325)*

- struct **DDS_ReliableWriterCacheEventCount**

  <<**eXtension**>> (p. 199) *The number of times the number of unacknowledged samples in the cache of a reliable writer hit a certain well-defined threshold.*

- struct **DDS_ReliableWriterCacheChangedStatus**

  <<**eXtension**>> (p. 199) *A summary of the state of a data writer's cache of unacknowledged samples written.*

- struct **DDS_ReliableReaderActivityChangedStatus**

  <<**eXtension**>> (p. 199) *Describes the activity (i.e. are acknowledgements forthcoming) of reliable readers matched to a reliable writer.*

- struct **DDS_DataWriterCacheStatus**

<<**eXtension**>> (p. 199) *The status of the writer's cache.*

ˆ struct **DDS_DataWriterProtocolStatus**

<<**eXtension**>> (p. 199) *The status of a writer's internal protocol related metrics, like the number of samples pushed, pulled, filtered; and status of wire protocol traffic.*

ˆ struct **DDS_DataWriterQos**

QoS policies supported by a ***DDSDataWriter*** *(*p. *1113) entity.*

5.14.1 Detailed Description

DDSDataWriter (p. 1113) entity and associated elements

5.15 Flow Controllers

<<*eXtension*>> (p. *199*) **DDSFlowController** (p. 1259) and associated elements

Classes

ˆ class **DDSFlowController**

<<**interface**>> (p. 199) *A flow controller is the object responsible for shaping the network traffic by determining when attached asynchronous* ***DDSDataWriter*** (p. *1113*) *instances are allowed to write data.*

ˆ struct **DDS_FlowControllerTokenBucketProperty_t**

DDSFlowController (p. *1259*) *uses the popular token bucket approach for open loop network flow control. The flow control characteristics are determined by the token bucket properties.*

ˆ struct **DDS_FlowControllerProperty_t**

Determines the flow control characteristics of the ***DDSFlowController*** (p. *1259*).

Enumerations

ˆ enum **DDS_FlowControllerSchedulingPolicy** {
DDS_RR_FLOW_CONTROLLER_SCHED_POLICY,
DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY,
DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY }

Kinds of flow controller scheduling policy.

Variables

ˆ char ∗ **DDS_DEFAULT_FLOW_CONTROLLER_NAME**

[**default**] *Special value of* ***DDS_PublishModeQosPolicy::flow_controller_name*** (p. *855*) *that refers to the built-in default flow controller.*

ˆ char ∗ **DDS_FIXED_RATE_FLOW_CONTROLLER_NAME**

Special value of ***DDS_PublishModeQosPolicy::flow_controller_name*** (p. *855*) *that refers to the built-in fixed-rate flow controller.*

ˆ char ∗ **DDS_ON_DEMAND_FLOW_CONTROLLER_NAME**

Special value of ***DDS_PublishModeQosPolicy::flow_controller_name*** *(p. 855) that refers to the built-in on-demand flow controller.*

5.15.1 Detailed Description

<<***eXtension***>> *(p. 199)* **DDSFlowController** (p. 1259) and associated elements

DDSFlowController (p. 1259) provides the network traffic shaping capability to asynchronous **DDSDataWriter** (p. 1113) instances. For use cases and advantages of publishing asnychronously, please refer to **DDS_PublishModeQosPolicy** (p. 853) of **DDS_DataWriterQos** (p. 553).

See also:

DDS_PublishModeQosPolicy (p. 853)
DDS_DataWriterQos::publish_mode (p. 558)
DDS_AsynchronousPublisherQosPolicy (p. 466)

5.15.2 Enumeration Type Documentation

5.15.2.1 enum DDS_FlowControllerSchedulingPolicy

Kinds of flow controller scheduling policy.

Samples written by an asynchronous **DDSDataWriter** (p. 1113) are not sent in the context of the **FooDataWriter::write** (p. 1484) call. Instead, the middleware puts the samples in a queue for future processing. The **DDSFlowController** (p. 1259) associated with each asynchronous DataWriter instance determines when the samples are actually sent.

Each **DDSFlowController** (p. 1259) maintains a separate FIFO queue for each unique destination (remote application). Samples written by asynchronous **DDSDataWriter** (p. 1113) instances associated with the flow controller, are placed in the queues that correspond to the intended destinations of the sample.

When tokens become available, a flow controller must decide which queue(s) to grant tokens first. This is determined by the flow controller's scheduling policy. Once a queue has been granted tokens, it is serviced by the asynchronous publishing thread. The queued up samples will be coalesced and sent to the corresponding destination. The number of samples sent depends on the data size and the number of tokens granted.

QoS:

DDS_FlowControllerProperty_t (p. 749)

Enumerator:

DDS_RR_FLOW_CONTROLLER_SCHED_POLICY Indicates to flow control in a round-robin fashion.

Whenever tokens become available, the flow controller distributes the tokens uniformly across all of its (non-empty) destination queues. No destinations are prioritized. Instead, all destinations are treated equally and are serviced in a round-robin fashion.

DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY Indicates to flow control in an earliest-deadline-first fashion.

A sample's deadline is determined by the time it was written plus the latency budget of the DataWriter at the time of the write call (as specified in the **DDS_LatencyBudgetQosPolicy** (p. 771)). The relative priority of a flow controller's destination queue is determined by the earliest deadline across all samples it contains.

When tokens become available, the **DDSFlowController** (p. 1259) distributes tokens to the destination queues in order of their deadline priority. In other words, the queue containing the sample with the earliest deadline is serviced first. The number of tokens granted equals the number of tokens required to send the first sample in the queue. Note that the priority of a queue may change as samples are sent (i.e. removed from the queue). If a sample must be sent to multiple destinations or two samples have an equal deadline value, the corresponding destination queues are serviced in a round-robin fashion.

Hence, under the default **DDS_LatencyBudgetQosPolicy::duration** (p. 772) setting, an `EDF_FLOW_CONTROLLER_SCHED_POLICY` **DDSFlowController** (p. 1259) preserves the order in which the user calls **FooDataWriter::write** (p. 1484) across the DataWriters associated with the flow controller.

Since the **DDS_LatencyBudgetQosPolicy** (p. 771) is mutable, a sample written second may contain an earlier deadline than the sample written first if the **DDS_LatencyBudgetQosPolicy::duration** (p. 772) value is sufficiently decreased in between writing the two samples. In that case, if the first sample is not yet written (still in queue waiting for its turn), it inherits the priority corresponding to the (earlier) deadline from the second sample.

In other words, the priority of a destination queue is always determined by the earliest deadline among all samples contained in the queue. This priority inheritance approach is required in order to both honor the updated **DDS_LatencyBudgetQosPolicy::duration** (p. 772) and adhere to the **DDSDataWriter** (p. 1113) in-order data delivery guarantee.

[**default**] for **DDSDataWriter** (p. 1113)

DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY Indicates /o flow control in an highest-priority-first fashion.

Determines the next destination queue to service as determined by the publication priority of the **DDSDataWriter** (p. 1113), channel of multi-channel DataWriter, or individual sample.

The relative priority of a flow controller's destination queue is determined by the highest publication priority of all samples it contains.

When tokens become available, the **DDSFlowController** (p. 1259) distributes tokens to the destination queues in order of their publication priority. In other words, the queue containing the sample with the highest publication priority is serviced first. The number of tokens granted equals the number of tokens required to send the first sample in the queue. Note that the priority of a queue may change as samples are sent (i.e. removed from the queue). If a sample must be sent to multiple destinations or two samples have an equal publication priority, the corresponding destination queues are serviced in a round-robin fashion.

This priority inheritance approach is required in order to both honor the designated publication priority and adhere to the **DDSDataWriter** (p. 1113) in-order data delivery guarantee.

5.15.3 Variable Documentation

5.15.3.1 char* DDS_DEFAULT_FLOW_CONTROLLER_NAME

[**default**] Special value of **DDS_PublishModeQosPolicy::flow_controller_name** (p. 855) that refers to the built-in default flow controller.

RTI Connext provides several built-in **DDSFlowController** (p. 1259) for use with an asynchronous **DDSDataWriter** (p. 1113). The user can choose to use the built-in flow controllers and optionally modify their properties or can create a custom flow controller.

By default, flow control is disabled. That is, the built-in **DDS_DEFAULT_FLOW_CONTROLLER_NAME** (p. 91) flow controller does not apply any flow control. Instead, it allows data to be sent asynchronously as soon as it is written by the **DDSDataWriter** (p. 1113).

Essentially, this is equivalent to a user-created **DDSFlowController** (p. 1259) with the following **DDS_FlowControllerProperty_t** (p. 749) settings:

- **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) = **DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `max_tokens` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `tokens_added_per_period` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `tokens_leaked_per_period` = 0

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `period` = 1 second

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `bytes_per_token` = **DDS_LENGTH_UNLIMITED** (p. 371)

See also:

DDSPublisher::create_datawriter (p. 1355)
DDSDomainParticipant::lookup_flowcontroller (p. 1186)
DDSFlowController::set_property (p. 1260)
DDS_PublishModeQosPolicy (p. 853)
DDS_AsynchronousPublisherQosPolicy (p. 466)

5.15.3.2 char* DDS_FIXED_RATE_FLOW_CONTROLLER_NAME

Special value of **DDS_PublishModeQosPolicy::flow_controller_name** (p. 855) that refers to the built-in fixed-rate flow controller.

RTI Connext provides several builtin **DDSFlowController** (p. 1259) for use with an asynchronous **DDSDataWriter** (p. 1113). The user can choose to use the built-in flow controllers and optionally modify their properties or can create a custom flow controller.

The built-in **DDS_FIXED_RATE_FLOW_CONTROLLER_NAME** (p. 92) flow controller shapes the network traffic by allowing data to be sent only once every second. Any accumulated samples destined for the same destination are coalesced into as few network packets as possible.

Essentially, this is equivalent to a user-created **DDSFlowController** (p. 1259) with the following **DDS_FlowControllerProperty_t** (p. 749) settings:

- **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) = **DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `max_tokens` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `tokens_added_per_period` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `tokens_leaked_per_period` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `period` = 1 second

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `bytes_per_token` = **DDS_LENGTH_UNLIMITED** (p. 371)

See also:

DDSPublisher::create_datawriter (p. 1355)
DDSDomainParticipant::lookup_flowcontroller (p. 1186)
DDSFlowController::set_property (p. 1260)
DDS_PublishModeQosPolicy (p. 853)
DDS_AsynchronousPublisherQosPolicy (p. 466)

5.15.3.3 char* DDS_ON_DEMAND_FLOW_CONTROLLER_NAME

Special value of **DDS_PublishModeQosPolicy::flow_controller_name** (p. 855) that refers to the built-in on-demand flow controller.

RTI Connext provides several builtin **DDSFlowController** (p. 1259) for use with an asynchronous **DDSDataWriter** (p. 1113). The user can choose to use the built-in flow controllers and optionally modify their properties or can create a custom flow controller.

The built-in **DDS_ON_DEMAND_FLOW_CONTROLLER_NAME** (p. 93) allows data to be sent only when the user calls **DDSFlowController::trigger_flow** (p. 1261). With each trigger, all accumulated data since the previous trigger is sent (across all **DDSPublisher** (p. 1346) or **DDSDataWriter** (p. 1113) instances). In other words, the network traffic shape is fully controlled by the user. Any accumulated samples destined for the same destination are coalesced into as few network packets as possible.

This external trigger source is ideal for users who want to implement some form of closed-loop flow control or who want to only put data on the wire every so many samples (e.g. with the number of samples based on **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525)).

Essentially, this is equivalent to a user-created **DDSFlowController** (p. 1259) with the following **DDS_FlowControllerProperty_t** (p. 749) settings:

- **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) = **DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `max_tokens` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `tokens_added_per_period` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `tokens_leaked_per_period` = **DDS_LENGTH_UNLIMITED** (p. 371)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `period` = **DDS_DURATION_INFINITE** (p. 305)

- **DDS_FlowControllerProperty_t::token_bucket** (p. 750) `bytes_per_token` = **DDS_LENGTH_UNLIMITED** (p. 371)

See also:

DDSPublisher::create_datawriter (p. 1355)
DDSDomainParticipant::lookup_flowcontroller (p. 1186)
DDSFlowController::trigger_flow (p. 1261)
DDSFlowController::set_property (p. 1260)
DDS_PublishModeQosPolicy (p. 853)
DDS_AsynchronousPublisherQosPolicy (p. 466)

5.16 Subscription Module

Contains the **DDSSubscriber** (p. 1390), **DDSDataReader** (p. 1087), **DDSReadCondition** (p. 1374), and **DDSQueryCondition** (p. 1372) classes, as well as the **DDSSubscriberListener** (p. 1414) and **DDSDataReaderListener** (p. 1108) interfaces, and more generally, all that is needed on the subscription side.

Modules

- **Subscribers**

  DDSSubscriber *(p. 1390) entity and associated elements*

- **DataReaders**

  DDSDataReader *(p. 1087) entity and associated elements*

- **Data Samples**

  DDS_SampleInfo *(p. 912),* ***DDS_SampleStateKind*** *(p. 111),* ***DDS_ViewStateKind*** *(p. 113),* ***DDS_InstanceStateKind*** *(p. 116) and associated elements*

5.16.1 Detailed Description

Contains the **DDSSubscriber** (p. 1390), **DDSDataReader** (p. 1087), **DDSReadCondition** (p. 1374), and **DDSQueryCondition** (p. 1372) classes, as well as the **DDSSubscriberListener** (p. 1414) and **DDSDataReaderListener** (p. 1108) interfaces, and more generally, all that is needed on the subscription side.

5.16.2 Access to data samples

Data is made available to the application by the following operations on **DDSDataReader** (p. 1087) objects: **FooDataReader::read** (p. 1447), **FooDataReader::read_w_condition** (p. 1454), **FooDataReader::take** (p. 1448), **FooDataReader::take_w_condition** (p. 1456), and the other variants of read() and take().

The general semantics of the `read()` operation is that the application only gets access to the corresponding data (i.e. a precise instance value); the data remains the responsibility of RTI Connext and can be read again.

The semantics of the `take()` operations is that the application takes full responsibility for the data; that data will no longer be available locally to RTI

Connext. Consequently, it is possible to access the same information multiple times only if all previous accesses were read() operations, not take().

Each of these operations returns a collection of `Data` values and associated **DDS_SampleInfo** (p. 912) objects. Each data value represents an atom of data information (i.e., a value for one instance). This collection may contain samples related to the same or different instances (identified by the key). Multiple samples can refer to the same instance if the settings of the **HISTORY** (p. 367) QoS allow for it.

To return the memory back to the middleware, every read() or take() that retrieves a sequence of samples must be followed with a call to FooDataReader::return_loan (p. 1471).

See also:

Interpretation of the SampleInfo (p. 913)

5.16.2.1 Data access patterns

The application accesses data by means of the operations `read` or `take` on the **DDSDataReader** (p. 1087). These operations return an ordered collection of `DataSamples` consisting of a **DDS_SampleInfo** (p. 912) part and a `Data` part.

The way RTI Connext builds the collection depends on QoS policies set on the **DDSDataReader** (p. 1087) and **DDSSubscriber** (p. 1390), as well as the `source_timestamp` of the samples, and the parameters passed to the `read()` / `take()` operations, namely:

- the desired sample states (any combination of **DDS_SampleStateKind** (p. 111))
- the desired view states (any combination of **DDS_ViewStateKind** (p. 113))
- the desired instance states (any combination of **DDS_InstanceStateKind** (p. 116))

The `read()` and `take()` operations are non-blocking and just deliver what is currently available that matches the specified states.

The `read_w_condition()` and `take_w_condition()` operations take a **DDSReadCondition** (p. 1374) object as a parameter instead of sample, view or instance states. The behaviour is that the samples returned will only be those for which the condition is **DDS_BOOLEAN_TRUE** (p. 298). These operations, in conjunction with **DDSReadCondition** (p. 1374) objects and a **DDSWaitSet** (p. 1433), allow performing waiting reads.

Once the data samples are available to the data readers, they can be read or taken by the application. The basic rule is that the application may do this in any order it wishes. This approach is very flexible and allows the application ultimate control.

To access data coherently, or in order, the **PRESENTATION** (p. 351) QoS must be set properly.

5.17 Subscribers

DDSSubscriber (p. 1390) entity and associated elements

Classes

- class **DDSSubscriberListener**

  <<**interface**>> (p. 199) ***DDSListener*** *(p. 1318) for status about a subscriber.*

- class **DDSSubscriberSeq**

  Declares IDL `sequence` < ***DDSSubscriber*** *(p. 1390)* > .

- class **DDSSubscriber**

  <<**interface**>> (p. 199) *A subscriber is the object responsible for actually receiving data from a subscription.*

- struct **DDS_SubscriberQos**

  QoS policies supported by a ***DDSSubscriber*** *(p. 1390) entity.*

Variables

- struct **DDS_DataReaderQos DDS_DATAREADER_QOS_DEFAULT**

  Special value for creating data reader with default QoS.

- struct **DDS_DataReaderQos DDS_DATAREADER_QOS_USE_TOPIC_QOS**

  Special value for creating ***DDSDataReader*** *(p. 1087) with a combination of the default* ***DDS_DataReaderQos*** *(p. 515) and the* ***DDS_TopicQos*** *(p. 965).*

5.17.1 Detailed Description

DDSSubscriber (p. 1390) entity and associated elements

5.17.2 Variable Documentation

5.17.2.1 struct DDS_DataReaderQos DDS_DATAREADER_QOS_DEFAULT

Special value for creating data reader with default QoS.

When used in **DDSSubscriber::create_datareader** (p. 1399), this special value is used to indicate that the **DDSDataReader** (p. 1087) should be created with the default **DDSDataReader** (p. 1087) QoS by means of the operation get_default_datareader_qos and using the resulting QoS to create the **DDSDataReader** (p. 1087).

When used in **DDSSubscriber::set_default_datareader_qos** (p. 1395), this special value is used to indicate that the default QoS should be reset back to the initial value that would be used if the **DDSSubscriber::set_default_datareader_qos** (p. 1395) operation had never been called.

When used in **DDSDataReader::set_qos** (p. 1102), this special value is used to indicate that the QoS of the **DDSDataReader** (p. 1087) should be changed to match the current default QoS set in the **DDSSubscriber** (p. 1390) that the **DDSDataReader** (p. 1087) belongs to.

Note: You cannot use this value to *get* the default QoS values for a DataReader; for this purpose, use **DDSDomainParticipant::get_default_datareader_qos** (p. 1152).

See also:

DDSSubscriber::create_datareader (p. 1399)
DDSSubscriber::set_default_datareader_qos (p. 1395)
DDSDataReader::set_qos (p. 1102)

Examples:

HelloWorld_subscriber.cxx.

5.17.2.2 struct DDS_DataReaderQos DDS_DATAREADER_QOS_USE_TOPIC_QOS

Special value for creating **DDSDataReader** (p. 1087) with a combination of the default **DDS_DataReaderQos** (p. 515) and the **DDS_TopicQos** (p. 965).

The use of this value is equivalent to the application obtaining the default **DDS_DataReaderQos** (p. 515) and the **DDS_TopicQos** (p. 965) (by means of the operation **DDSTopic::get_qos** (p. 1423)) and then combining these two QoS using the operation **DDSSubscriber::copy_from_topic_qos** (p. 1409) whereby any policy that is set on the **DDS_TopicQos** (p. 965) "overrides"

the corresponding policy on the default QoS. The resulting QoS is then applied to the creation of the **DDSDataReader** (p. 1087).

This value should only be used in **DDSSubscriber::create_datareader** (p. 1399).

See also:

DDSSubscriber::create_datareader (p. 1399)
DDSSubscriber::get_default_datareader_qos (p. 1394)
DDSTopic::get_qos (p. 1423)
DDSSubscriber::copy_from_topic_qos (p. 1409)

5.18 DataReaders

DDSDataReader (p. 1087) entity and associated elements

Modules

- **Read Conditions**

  DDSReadCondition *(p. 1374) and associated elements*

- **Query Conditions**

  DDSQueryCondition *(p. 1372) and associated elements*

Classes

- struct **FooDataReader**

  <<**interface**>> (p. 199) <<**generic**>> (p. 199) *User data type-specific data reader.*

- class **DDSDataReaderSeq**

  Declares IDL `sequence` *<* ***DDSDataReader*** *(p. 1087) > .*

- class **DDSDataReaderListener**

  <<**interface**>> (p. 199) ***DDSListener*** *(p. 1318) for reader status.*

- class **DDSDataReader**

  <<**interface**>> (p. 199) *Allows the application to: (1) declare the data it wishes to receive (i.e. make a subscription) and (2) access the data received by the attached* ***DDSSubscriber*** *(p. 1390).*

- struct **DDS_RequestedDeadlineMissedStatus**

  DDS_REQUESTED_DEADLINE_MISSED_STATUS *(p. 323)*

- struct **DDS_LivelinessChangedStatus**

  DDS_LIVELINESS_CHANGED_STATUS *(p. 325)*

- struct **DDS_RequestedIncompatibleQosStatus**

  DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS *(p. 323)*

- struct **DDS_SampleLostStatus**

  DDS_SAMPLE_LOST_STATUS *(p. 324)*

ˆ struct **DDS_SampleRejectedStatus**

DDS_SAMPLE_REJECTED_STATUS *(p. 324)*

ˆ struct **DDS_SubscriptionMatchedStatus**

DDS_SUBSCRIPTION_MATCHED_STATUS *(p. 326)*

ˆ struct **DDS_DataReaderCacheStatus**

<<**eXtension**>> (p. 199) *The status of the reader's cache.*

ˆ struct **DDS_DataReaderProtocolStatus**

<<**eXtension**>> (p. 199) *The status of a reader's internal protocol related metrics, like the number of samples received, filtered, rejected; and status of wire protocol traffic.*

ˆ struct **DDS_DataReaderQos**

QoS policies supported by a ***DDSDataReader*** *(p. 1087) entity.*

Enumerations

ˆ enum **DDS_SampleLostStatusKind** {

DDS_NOT_LOST,

DDS_LOST_BY_WRITER,

DDS_LOST_BY_INSTANCES_LIMIT,

DDS_LOST_BY_REMOTE_WRITERS_PER_INSTANCE_-LIMIT,

DDS_LOST_BY_INCOMPLETE_COHERENT_SET,

DDS_LOST_BY_LARGE_COHERENT_SET,

DDS_LOST_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT,

DDS_LOST_BY_VIRTUAL_WRITERS_LIMIT,

DDS_LOST_BY_REMOTE_WRITERS_PER_SAMPLE_LIMIT,

DDS_LOST_BY_AVAILABILITY_WAITING_TIME,

DDS_LOST_BY_REMOTE_WRITER_SAMPLES_PER_-VIRTUAL_QUEUE_LIMIT,

DDS_LOST_BY_OUT_OF_MEMORY }

Kinds of reasons why a sample was lost.

^ enum **DDS_SampleRejectedStatusKind** {

DDS_NOT_REJECTED,

DDS_REJECTED_BY_INSTANCES_LIMIT,

DDS_REJECTED_BY_SAMPLES_LIMIT,

DDS_REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT,

DDS_REJECTED_BY_REMOTE_WRITERS_LIMIT,

DDS_REJECTED_BY_REMOTE_WRITERS_PER_-INSTANCE_LIMIT,

DDS_REJECTED_BY_SAMPLES_PER_REMOTE_WRITER_-LIMIT,

DDS_REJECTED_BY_VIRTUAL_WRITERS_LIMIT,

DDS_REJECTED_BY_REMOTE_WRITERS_PER_SAMPLE_-LIMIT,

DDS_REJECTED_BY_REMOTE_WRITER_SAMPLES_PER_-VIRTUAL_QUEUE_LIMIT }

Kinds of reasons for rejecting a sample.

5.18.1 Detailed Description

DDSDataReader (p. 1087) entity and associated elements

5.18.2 Enumeration Type Documentation

5.18.2.1 enum DDS_SampleLostStatusKind

Kinds of reasons why a sample was lost.

Enumerator:

DDS_NOT_LOST The sample was not lost.

See also:

ResourceLimitsQosPolicy

DDS_LOST_BY_WRITER A DataWriter removed the sample before being received by the **DDSDataReader** (p. 1087).

This constant is an extension to the DDS standard.

DDS_LOST_BY_INSTANCES_LIMIT A resource limit on the number of instances was reached.

This constant is an extension to the DDS standard.

See also:

ResourceLimitsQosPolicy

DDS_LOST_BY_REMOTE_WRITERS_PER_INSTANCE_LIMIT
A resource limit on the number of remote writers for a single instance from which a **DDSDataReader** (p. 1087) may read was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_LOST_BY_INCOMPLETE_COHERENT_SET A sample is lost because it is part of an incomplete coherent set.

This constant is an extension to the DDS standard.

DDS_LOST_BY_LARGE_COHERENT_SET A sample is lost because it is part of a large coherent set.

This constant is an extension to the DDS standard.

DDS_LOST_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT
A resource limit on the number of samples from a given remote writer that a **DDSDataReader** (p. 1087) may store was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_LOST_BY_VIRTUAL_WRITERS_LIMIT A resource limit on the number of virtual writers from which a **DDSDataReader** (p. 1087) may read was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_LOST_BY_REMOTE_WRITERS_PER_SAMPLE_LIMIT
A resource limit on the number of remote writers per sample was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_LOST_BY_AVAILABILITY_WAITING_TIME **DDS_AvailabilityQosPolicy::max_data_availability_waiting_time** (p. 473) expired.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_LOST_BY_REMOTE_WRITER_SAMPLES_PER_VIRTUAL_QUEUE_LIMIT A resource limit on the number of samples published by a remote writer on behalf of a virtual writer that a **DDSDataReader** (p. 1087) may store was reached.

This constant is an extension to the DDS standard.

See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_LOST_BY_OUT_OF_MEMORY A sample was lost because there was not enough memory to store the sample.

This constant is an extension to the DDS standard.

See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

5.18.2.2 enum DDS_SampleRejectedStatusKind

Kinds of reasons for rejecting a sample.

Enumerator:

DDS_NOT_REJECTED Samples are never rejected.

See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_INSTANCES_LIMIT A resource limit on the number of instances was reached.

See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_SAMPLES_LIMIT A resource limit on the number of samples was reached.

See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT A resource limit on the number of samples per instance was reached.

See also:

ResourceLimitsQosPolicy

DDS_REJECTED_BY_REMOTE_WRITERS_LIMIT A resource limit on the number of remote writers from which a **DDSDataReader** (p. 1087) may read was reached.

This constant is an extension to the DDS standard.

See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_REMOTE_WRITERS_PER_INSTANCE_LIMIT
A resource limit on the number of remote writers for a single instance from which a **DDSDataReader** (p. 1087) may read was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT
A resource limit on the number of samples from a given remote writer that a **DDSDataReader** (p. 1087) may store was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_VIRTUAL_WRITERS_LIMIT A resource limit on the number of virtual writers from which a **DDSDataReader** (p. 1087) may read was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_REMOTE_WRITERS_PER_SAMPLE_LIMIT
A resource limit on the number of remote writers per sample was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

DDS_REJECTED_BY_REMOTE_WRITER_SAMPLES_PER_VIRTUAL_QUEUE_LIM
A resource limit on the number of samples published by a remote writer on behalf of a virtual writer that a **DDSDataReader** (p. 1087) may store was reached.

This constant is an extension to the DDS standard.
See also:

DDS_DataReaderResourceLimitsQosPolicy (p. 521)

5.19 Read Conditions

DDSReadCondition (p. 1374) and associated elements

Classes

- ˆ class **DDSReadCondition**

  <<**interface**>> (p. 199) *Conditions specifically dedicated to read operations and attached to one* ***DDSDataReader*** *(p. 1087).*

5.19.1 Detailed Description

DDSReadCondition (p. 1374) and associated elements

5.20 Query Conditions

DDSQueryCondition (p. 1372) and associated elements

Classes

- class **DDSQueryCondition**

  <<**interface**>> (p. 199) *These are specialised* ***DDSReadCondition*** *(p. 1374) objects that allow the application to also specify a filter on the locally available data.*

5.20.1 Detailed Description

DDSQueryCondition (p. 1372) and associated elements

5.21 Data Samples

DDS_SampleInfo (p. 912), **DDS_SampleStateKind** (p. 111), **DDS_ViewStateKind** (p. 113), **DDS_InstanceStateKind** (p. 116) and associated elements

Modules

- **Sample States**

  DDS_SampleStateKind *(p. 111) and associated elements*

- **View States**

  DDS_ViewStateKind *(p. 113) and associated elements*

- **Instance States**

  DDS_InstanceStateKind *(p. 116) and associated elements*

Classes

- struct **DDS_SampleInfo**

  Information that accompanies each sample that is `read` *or* `taken`.

- struct **DDS_SampleInfoSeq**

  Declares IDL `sequence` < ***DDS_SampleInfo*** *(p. 912)* > .

Functions

- void **DDS_SampleInfo_get_sample_identity** (const struct **DDS_SampleInfo** *self, struct **DDS_SampleIdentity_t** *identity)

  <<**eXtension**>> (p. 199) *Retrieves the identity of the sample*

- void **DDS_SampleInfo_get_related_sample_identity** (const struct **DDS_SampleInfo** *self, struct **DDS_SampleIdentity_t** *related_identity)

  <<**eXtension**>> (p. 199) *Retrieves the identity of a sample related to this one*

5.21.1 Detailed Description

DDS_SampleInfo (p. 912), **DDS_SampleStateKind** (p. 111), **DDS_ViewStateKind** (p. 113), **DDS_InstanceStateKind** (p. 116) and associated elements

5.21.2 Function Documentation

5.21.2.1 void DDS_SampleInfo_get_sample_identity (const struct DDS_SampleInfo * *self*, struct DDS_SampleIdentity_t * *identity*)

<<*eXtension*>> *(p. 199)* Retrieves the identity of the sample

The identity is composed of the **DDS_SampleInfo::original_publication_virtual_guid** (p. 920) and the **DDS_SampleInfo::original_publication_virtual_sequence_number** (p. 920)

See also:

DDS_SampleInfo_get_related_sample_identity (p. 110)

5.21.2.2 void DDS_SampleInfo_get_related_sample_identity (const struct DDS_SampleInfo * *self*, struct DDS_SampleIdentity_t * *related_identity*)

<<*eXtension*>> *(p. 199)* Retrieves the identity of a sample related to this one

A sample can be logically related to another sample, when the **DDS-DataWriter** (p. 1113) wrote it using **DDS_WriteParams_t::related_sample_identity** (p. 1068). By default, a sample does not have a related sample, and this operation returns **DDS_UNKNOWN_SAMPLE_IDENTITY** (p. 449).

The related identity is composed of the **DDS_SampleInfo::related_original_publication_virtual_guid** (p. 921) and the **DDS_SampleInfo::related_original_publication_virtual_sequence_number** (p. 921)

See also:

DDS_WriteParams_t::related_sample_identity (p. 1068)

5.22 Sample States

DDS_SampleStateKind (p. 111) and associated elements

Typedefs

- typedef **DDS_UnsignedLong DDS_SampleStateMask**

  A bit-mask (list) of sample states, i.e. ***DDS_SampleStateKind*** *(p. 111).*

Enumerations

- enum **DDS_SampleStateKind** {
  DDS_READ_SAMPLE_STATE = 0x0001 << 0,
  DDS_NOT_READ_SAMPLE_STATE = 0x0001 << 1 }

  Indicates whether or not a sample has ever been read.

Variables

- const **DDS_SampleStateMask DDS_ANY_SAMPLE_STATE**

  Any sample state ***DDS_READ_SAMPLE_STATE*** *(p. 112) |* ***DDS_NOT_READ_SAMPLE_STATE*** *(p. 112).*

5.22.1 Detailed Description

DDS_SampleStateKind (p. 111) and associated elements

5.22.2 Typedef Documentation

5.22.2.1 typedef DDS_UnsignedLong DDS_SampleStateMask

A bit-mask (list) of sample states, i.e. **DDS_SampleStateKind** (p. 111).

5.22.3 Enumeration Type Documentation

5.22.3.1 enum DDS_SampleStateKind

Indicates whether or not a sample has ever been read.

For each sample received, the middleware internally maintains a sample_state relative to each **DDSDataReader** (p. 1087). The sample state can be either:

- **DDS_READ_SAMPLE_STATE** (p. 112) indicates that the **DDSDataReader** (p. 1087) has already accessed that sample by means of a read or take operation.

- **DDS_NOT_READ_SAMPLE_STATE** (p. 112) indicates that the **DDSDataReader** (p. 1087) has not accessed that sample before.

The sample state will, in general, be different for each sample in the collection returned by read or take.

Enumerator:

DDS_READ_SAMPLE_STATE Sample has been read.

DDS_NOT_READ_SAMPLE_STATE Sample has not been read.

5.22.4 Variable Documentation

5.22.4.1 const DDS_SampleStateMask DDS_ANY_SAMPLE_STATE

Any sample state **DDS_READ_SAMPLE_STATE** (p. 112) | **DDS_NOT_READ_SAMPLE_STATE** (p. 112).

Examples:

HelloWorld_subscriber.cxx.

5.23 View States

DDS_ViewStateKind (p. 113) and associated elements

Typedefs

- typedef **DDS_UnsignedLong DDS_ViewStateMask**

  A bit-mask (list) of view states, i.e. ***DDS_ViewStateKind*** *(p. 113).*

Enumerations

- enum **DDS_ViewStateKind** {
  DDS_NEW_VIEW_STATE = 0x0001 << 0,
  DDS_NOT_NEW_VIEW_STATE = 0x0001 << 1 }

  Indicates whether or not an instance is new.

Variables

- const **DDS_ViewStateMask DDS_ANY_VIEW_STATE**

  Any view state ***DDS_NEW_VIEW_STATE*** *(p. 114) |* ***DDS_NOT_NEW_VIEW_STATE*** *(p. 114).*

5.23.1 Detailed Description

DDS_ViewStateKind (p. 113) and associated elements

5.23.2 Typedef Documentation

5.23.2.1 typedef DDS_UnsignedLong DDS_ViewStateMask

A bit-mask (list) of view states, i.e. **DDS_ViewStateKind** (p. 113).

5.23.3 Enumeration Type Documentation

5.23.3.1 enum DDS_ViewStateKind

Indicates whether or not an instance is new.

For each instance (identified by the key), the middleware internally maintains a view state relative to each **DDSDataReader** (p. 1087). The view state can be either:

- **DDS_NEW_VIEW_STATE** (p. 114) indicates that either this is the first time that the **DDSDataReader** (p. 1087) has ever accessed samples of that instance, or else that the **DDSDataReader** (p. 1087) has accessed previous samples of the instance, but the instance has since been reborn (i.e. become not-alive and then alive again). These two cases are distinguished by examining the **DDS_SampleInfo::disposed_generation_count** (p. 918) and the **DDS_SampleInfo::no_writers_generation_count** (p. 918).

- **DDS_NOT_NEW_VIEW_STATE** (p. 114) indicates that the **DDSDataReader** (p. 1087) has already accessed samples of the same instance and that the instance has not been reborn since.

The view_state available in the **DDS_SampleInfo** (p. 912) is a snapshot of the view state of the instance relative to the **DDSDataReader** (p. 1087) used to access the samples at the time the collection was obtained (i.e. at the time read or take was called). The view_state is therefore the same for all samples in the returned collection that refer to the same instance.

Once an instance has been detected as not having any "live" writers and all the samples associated with the instance are "taken" from the **DDSDataReader** (p. 1087), the middleware can reclaim all local resources regarding the instance. Future samples will be treated as "never seen."

Enumerator:

DDS_NEW_VIEW_STATE New instance. This latest generation of the instance has not previously been accessed.

DDS_NOT_NEW_VIEW_STATE Not a new instance. This latest generation of the instance has previously been accessed.

5.23.4 Variable Documentation

5.23.4.1 const DDS_ViewStateMask DDS_ANY_VIEW_STATE

Any view state **DDS_NEW_VIEW_STATE** (p. 114) | **DDS_NOT_NEW_VIEW_STATE** (p. 114).

Examples:

HelloWorld_subscriber.cxx.

5.24 Instance States

DDS_InstanceStateKind (p. 116) and associated elements

Typedefs

- typedef **DDS_UnsignedLong DDS_InstanceStateMask**

  A bit-mask (list) of instance states, i.e. ***DDS_InstanceStateKind*** *(p. 116).*

Enumerations

- enum **DDS_InstanceStateKind** {

  DDS_ALIVE_INSTANCE_STATE = 0x0001 << 0,

  DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE = 0x0001 << 1,

  DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE = 0x0001 << 2 }

  Indicates is the samples are from a live ***DDSDataWriter*** *(p. 1113) or not.*

Variables

- const **DDS_InstanceStateMask DDS_ANY_INSTANCE_STATE**

  Any instance state ALIVE_INSTANCE_STATE | NOT_ALIVE_DISPOSED_INSTANCE_STATE | NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.

- const **DDS_InstanceStateMask DDS_NOT_ALIVE_INSTANCE_STATE**

  Not alive instance state NOT_ALIVE_DISPOSED_INSTANCE_STATE | NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.

5.24.1 Detailed Description

DDS_InstanceStateKind (p. 116) and associated elements

5.24.2 Typedef Documentation

5.24.2.1 typedef DDS_UnsignedLong DDS_InstanceStateMask

A bit-mask (list) of instance states, i.e. **DDS_InstanceStateKind** (p. 116).

5.24.3 Enumeration Type Documentation

5.24.3.1 enum DDS_InstanceStateKind

Indicates is the samples are from a live **DDSDataWriter** (p. 1113) or not.

For each instance, the middleware internally maintains an instance state. The instance state can be:

- **DDS_ALIVE_INSTANCE_STATE** (p. 117) indicates that (a) samples have been received for the instance, (b) there are live **DDSDataWriter** (p. 1113) entities writing the instance, and (c) the instance has not been explicitly disposed (or else more samples have been received after it was disposed).

- **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) indicates the instance was explicitly disposed by a **DDSDataWriter** (p. 1113) by means of the dispose operation.

- **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117) indicates the instance has been declared as not-alive by the **DDSDataReader** (p. 1087) because it detected that there are no live **DDSDataWriter** (p. 1113) entities writing that instance.

The precise behavior events that cause the instance state to change depends on the setting of the OWNERSHIP QoS:

- If **OWNERSHIP** (p. 355) is set to **DDS_EXCLUSIVE_OWNERSHIP_QOS** (p. 356), then the instance state becomes **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) only if the **DDSDataWriter** (p. 1113) that "owns" the instance explicitly disposes it. The instance state becomes **DDS_ALIVE_INSTANCE_STATE** (p. 117) again only if the **DDSDataWriter** (p. 1113) that owns the instance writes it.

- If **OWNERSHIP** (p. 355) is set to **DDS_SHARED_OWNERSHIP_QOS** (p. 356), then the instance state becomes **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) if any **DDSDataWriter**

(p. 1113) explicitly disposes the instance. The instance state becomes **DDS_ALIVE_INSTANCE_STATE** (p. 117) as soon as any **DDSDataWriter** (p. 1113) writes the instance again.

The instance state available in the **DDS_SampleInfo** (p. 912) is a snapshot of the instance state of the instance at the time the collection was obtained (i.e. at the time read or take was called). The instance state is therefore the same for all samples in the returned collection that refer to the same instance.

Enumerator:

DDS_ALIVE_INSTANCE_STATE Instance is currently in existence.

DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE Not alive disposed instance. The instance has been disposed by a DataWriter.

DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE Not alive no writers for instance. None of the **DDSDataWriter** (p. 1113) objects are currently alive (according to the **LIVELINESS** (p. 358)) are writing the instance.

5.24.4 Variable Documentation

5.24.4.1 const DDS_InstanceStateMask DDS_ANY_INSTANCE_STATE

Any instance state ALIVE_INSTANCE_STATE | NOT_ALIVE_DISPOSED_INSTANCE_STATE | NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.

Examples:

HelloWorld_subscriber.cxx.

5.24.4.2 const DDS_InstanceStateMask DDS_NOT_ALIVE_INSTANCE_STATE

Not alive instance state NOT_ALIVE_DISPOSED_INSTANCE_STATE | NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.

5.25 Infrastructure Module

Defines the abstract classes and the interfaces that are refined by the other modules. Contains common definitions such as return codes, status values, and QoS policies.

Modules

- **Time Support**

  Time and duration types and defines.

- **GUID Support**

  <<**eXtension**>> (p. 199) *GUID type and defines.*

- **Sequence Number Support**

  <<**eXtension**>> (p. 199) *Sequence number type and defines.*

- **Exception Codes**

  <<**eXtension**>> (p. 199) *Exception codes.*

- **Return Codes**

  Types of return codes.

- **Status Kinds**

  Kinds of communication status.

- **QoS Policies**

  Quality of Service (QoS) policies.

- **Entity Support**

  DDSEntity *(p. 1253),* ***DDSListener*** *(p. 1318) and related items.*

- **Conditions and WaitSets**

  DDSCondition *(p. 1075) and* ***DDSWaitSet*** *(p. 1433) and related items.*

- **WriteParams**

  <<**eXtension**>> (p. 199)

- **Octet Buffer Support**

  <<**eXtension**>> (p. 199) *Octet buffer creation, cloning, and deletion.*

- **Sequence Support**

*The **FooSeq** (p. 1494) interface allows you to work with variable-length collections of homogeneous data.*

- **String Support**

  <<**eXtension**>> (p. 199) *String creation, cloning, assignment, and deletion.*

5.25.1 Detailed Description

Defines the abstract classes and the interfaces that are refined by the other modules. Contains common definitions such as return codes, status values, and QoS policies.

5.26 Built-in Sequences

Defines sequences of primitive data type.

Classes

- struct **DDS_CharSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Char*** (p. 299) >.
- struct **DDS_WcharSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Wchar*** (p. 299) >.
- struct **DDS_OctetSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Octet*** (p. 299) >.
- struct **DDS_ShortSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Short*** (p. 299) >.
- struct **DDS_UnsignedShortSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_UnsignedShort*** (p. 299) >.
- struct **DDS_LongSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Long*** (p. 300) >.
- struct **DDS_UnsignedLongSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_UnsignedLong*** (p. 300) >.
- struct **DDS_LongLongSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_LongLong*** (p. 300) >.
- struct **DDS_UnsignedLongLongSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_UnsignedLongLong*** (p. 300) >.
- struct **DDS_FloatSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Float*** (p. 300) >.
- struct **DDS_DoubleSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_Double*** (p. 300) >.
- struct **DDS_LongDoubleSeq**
  Instantiates FooSeq (p. 1494) < ***DDS_LongDouble*** (p. 300) >.

- struct **DDS_BooleanSeq**

  Instantiates FooSeq (p. 1494) < ***DDS_Boolean*** *(p. 301)* >.

- struct **DDS_StringSeq**

  Instantiates FooSeq (p. 1494) < *char** > *with value type semantics.*

- struct **DDS_WstringSeq**

  Instantiates FooSeq (p. 1494) < ***DDS_Wchar*** *(p. 299)** >.

5.26.1 Detailed Description

Defines sequences of primitive data type.

5.27 Multi-channel DataWriters

APIs related to Multi-channel DataWriters.

5.27.1 What is a Multi-channel DataWriter?

A Multi-channel **DDSDataWriter** (p. 1113) is a **DDSDataWriter** (p. 1113) that is configured to send data over multiple multicast addresses, according to some filtering criteria applied to the data.

To determine which multicast addresses will be used to send the data, the middleware evaluates a set of filters that are configured for the **DDSDataWriter** (p. 1113). Each filter "guards" a channel (a set of multicast addresses). Each time a multi-channel **DDSDataWriter** (p. 1113) writes data, the filters are applied. If a filter evaluates to true, the data is sent over that filter's associated channel (set of multicast addresses). We refer to this type of filter as a Channel Guard filter.

5.27.2 Configuration on the Writer Side

To configure a multi-channel **DDSDataWriter** (p. 1113), simply define a list of all its channels in the **DDS_MultiChannelQosPolicy** (p. 796).

The **DDS_MultiChannelQosPolicy** (p. 796) is propagated along with discovery traffic. The value of this policy is available in **DDS_PublicationBuiltinTopicData::locator_filter** (p. 846).

5.27.3 Configuration on the Reader Side

No special changes are required in a subscribing application to get data from a multichannel **DDSDataWriter** (p. 1113). If you want the **DDSDataReader** (p. 1087) to subscribe to only a subset of the channels, use a **DDSContentFilteredTopic** (p. 1081).

For more information on Multi-channel DataWriters, refer to the `User's Manual`.

5.27.4 Reliability with Multi-Channel DataWriters

5.27.4.1 Reliable Delivery

Reliable delivery is only guaranteed when the **DDS_PresentationQosPolicy::access_scope** (p. 826) is set to **DDS_**

INSTANCE_PRESENTATION_QOS (p. 352) and the filters in **DDS_MultiChannelQosPolicy** (p. 796) are keyed-only based.

If any of the guard filters are based on non-key fields, RTI Connext only guarantees reception of the most recent data from the MultiChannel DataWriter.

5.27.4.2 Reliable Protocol Considerations

Reliability is maintained on a per-channel basis. Each channel has its own reliability channel send queue. The size of that queue is limited by **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) and/or **DDS_DataWriterResourceLimitsQosPolicy::max_batches** (p. 563).

The protocol parameters described in **DDS_DataWriterProtocolQosPolicy** (p. 535) are applied per channel, with the following exceptions:

DDS_RtpsReliableWriterProtocol_t::low_watermark (p. 892) and **DDS_RtpsReliableWriterProtocol_t::high_watermark** (p. 892): The low watermark and high watermark control the queue levels (in number of samples) that determine when to switch between regular and fast heartbeat rates. With MultiChannel DataWriters, high_watermark and low_watermark refer to the DataWriter's queue (not the reliability channel queue). Therefore, periodic heartbeating cannot be controlled on a per-channel basis.

Important: With MultiChannel DataWriters, low_watermark and high_watermark refer to application samples even if batching is enabled. This behavior differs from the one without MultiChannel DataWriters (where low_watermark and high_watermark refer to batches).

DDS_RtpsReliableWriterProtocol_t::heartbeats_per_max_samples (p. 896): This field defines the number of heartbeats per send queue. For MultiChannel DataWriters, the value is applied per channel. However, the send queue size that is used to calculate the a piggyback heartbeat rate is defined per DataWriter (see **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881))

Important: With MultiChannel DataWriters, heartbeats_per_max_samples refers to samples even if batching is enabled. This behavior differs from the one without MultiChannels DataWriters (where heartbeats_per_max_samples refers to batches).

With batching and MultiChannel DataWriters, the size of the DataWriter's send queue should be configured using **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) instead of max_batches **DDS_DataWriterResourceLimitsQosPolicy::max_batches** (p. 563) in order to take advantage of heartbeats_per_max_samples.

5.28 Pluggable Transports

APIs related to RTI Connext pluggable transports.

Modules

- **Using Transport Plugins**

  Configuring transports used by RTI Connext.

- **Built-in Transport Plugins**

  Transport plugins delivered with RTI Connext.

5.28.1 Detailed Description

APIs related to RTI Connext pluggable transports.

5.28.2 Overview

RTI Connext has a pluggable transports architecture. The core of RTI Connext is transport agnostic; it does not make any assumptions about the actual transports used to send and receive messages. Instead, the RTI Connext core uses an abstract "transport API" to interact with the **transport plugins** which implement that API.

A transport plugin implements the abstract transport API and performs the actual work of sending and receiving messages over a physical transport. A collection of **builtin plugins** (see **Built-in Transport Plugins** (p. 136)) is delivered with RTI Connext for commonly used transports. New transport plugins can easily be created, thus enabling RTI Connext applications to run over transports that may not even be conceived yet. This is a powerful capability and that distinguishes RTI Connext from competing middleware approaches.

RTI Connext also provides a set of APIs for installing and configuring transport plugins to be used in an application. So that RTI Connext applications work out of the box, a subset of the builtin transport plugins is *preconfigured* by default (see **DDS_TransportBuiltinQosPolicy** (p. 969)). You can "turn-off" some or all of the builtin transport plugins. In addition, you can configure other transport plugins for use by the application.

5.28.3 Transport Aliases

In order to use a transport plugin instance in an RTI Connext application, it must be registered with a **DDSDomainParticipant** (p. 1139). When you register a transport, you specify a sequence of "alias" strings to symbolically refer to the transport plugin. The same alias strings can be used to register more than one transport plugin.

You can register multiple transport plugins with a **DDSDomainParticipant** (p. 1139). An **alias** symbolically refers to one or more transport plugins registered with the **DDSDomainParticipant** (p. 1139). Builtin transport plugin instances can be referred to using preconfigured aliases (see **TRANSPORT_BUILTIN** (p. 396)).

A transport plugin's class name is automatically used as an implicit alias. It can be used to refer to all the transport plugin instances of that class.

You can use aliases to refer to transport plugins, in order to specify:

- the transport plugins to use for **discovery** (see **DDS_DiscoveryQosPolicy::enabled_transports** (p. 583)), and for **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) entities (see **DDS_TransportSelectionQosPolicy** (p. 985)).

- the **multicast** addresses on which to receive discovery messages (see **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584)), and the multicast addresses and ports on which to receive user data (see **DDS_DataReaderQos::multicast** (p. 520)).

- the **unicast ports** used for user data (see **DDS_TransportUnicastQosPolicy** (p. 987)) on both **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) entities.

- the transport plugins used to parse an address string in a locator (**Locator Format** (p. 389) and **NDDS_DISCOVERY_PEERS** (p. 388)).

A **DDSDomainParticipant** (p. 1139) (and contained its entities) start using a transport plugin after the **DDSDomainParticipant** (p. 1139) is enabled (see **DDSEntity::enable** (p. 1256)). An entity will use *all* the transport plugins that match the specified transport QoS policy. All transport plugins are treated uniformly, regardless of how they were created or registered; there is no notion of some transports being more "special" that others.

5.28.4 Transport Lifecycle

A transport plugin is owned by whoever created it. Thus, if you create and register a transport plugin with a **DDSDomainParticipant** (p. 1139), you are responsible for deleting it by calling its destructor. Note that builtin transport plugins (**TRANSPORT_BUILTIN** (p. 396)) and transport plugins that are

loaded through the **PROPERTY** (p. 436) QoS policy (see **Loading Transport Plugins through Property QoS Policy of Domain Participant** (p. 131)) are automatically managed by RTI Connext.

A user-created transport plugin must not be deleted while it is still in use by a **DDSDomainParticipant** (p. 1139). This generally means that a user-created transport plugin instance can only be deleted after the **DDSDomainParticipant** (p. 1139) with which it was registered is deleted (see **DDSDomainParticipantFactory::delete_participant** (p. 1236)). Note that a transport plugin *cannot* be "unregistered" from a **DDSDomainParticipant** (p. 1139).

A transport plugin instance cannot be registered with more than one **DDSDomainParticipant** (p. 1139) at a time. This requirement is necessary to guarantee the multi-threaded safety of the transport API.

If the same physical transport resources are to be used with more than one **DDSDomainParticipant** (p. 1139) in the same address space, the transport plugin should be written in such a way so that it can be instantiated multiple times—once for each **DDSDomainParticipant** (p. 1139) in the address space. Note that it is always possible to write the transport plugin so that multiple transport plugin instances share the same underlying resources; however the burden (if any) of guaranteeing multi-threaded safety to access shared resource shifts to the transport plugin developer.

5.28.5 Transport Class Attributes

A transport plugin instance is associated with two kinds of attributes:

- the *class* attributes that are decided by the plugin writer; these are invariant across all instances of the transport plugin class, and

- the *instance* attributes that can be set on a per instance basis by the transport plugin user.

Every transport plugin must specify the following class attributes.

transport class id (see NDDS_Transport_Property_t::classid (p. 1524))
: Identifies a transport plugin implementation class. It denotes a unique "class" to which the transport plugin instance belongs. The class is used to distinguish between different transport plugin implementations. Thus, a transport plugin vendor should ensure that its transport plugin implementation has a unique class.

  Two transport plugin instances report the same class *iff* they have compatible implementations. Transport plugin instances with mismatching classes are not allowed (by the RTI Connext Core) to communicate with one another.

  Multiple implementations (possibly from different vendors) for a physical

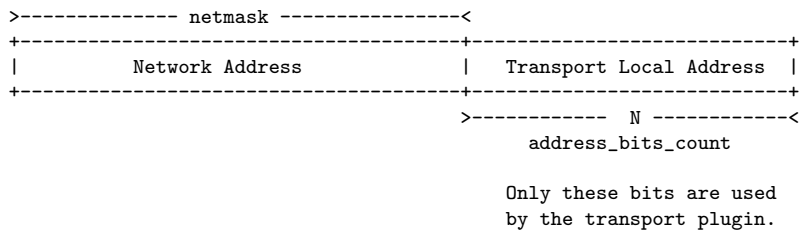
transport mechanism can co-exist in an RTI Connext application, provided they use different transport class IDs.

The class ID can also be used to distinguish between different transport protocols over the same physical transport network (e.g., UDP vs. TCP over the IP routing infrastructure).

transport significant address bit count (see **NDDS_Transport_Property_t::address_bit_count** (p. 1524))

RTI Connext's addressing is modeled after the IPv6 and uses 128-bit addresses (**Address** (p. 250)) to route messages.

A transport plugin is expected to map the transport's internal addressing scheme to 128-bit addresses. In general, this mapping is likely to use only `N` least significant bits (LSB); these are specified by this attribute.



The remaining bits of an address using the 128-bit address representation will be considered as part of the "network address" (see **Transport Network Address** (p. 128)) and thus ignored by the transport plugin's internal addressing scheme.

For *unicast* addresses, the transport plugin is expected to ignore the higher (128 - **NDDS_Transport_Property_t::address_bit_count** (p. 1524)) bits. RTI Connext is free to manipulate those bits freely in the addresses passed in/out to the transport plugin APIs.

Theoretically, the significant address bits count, `N` is related to the size of the underlying transport network as follows:

$$address\_bits\_count >= ceil(log_2(total\_addressable\_transport\_unicast\_interfaces))$$

The equality holds when the most compact (theoretical) internal address mapping scheme is used. A practical address mapping scheme may waste some bits.

5.28.6 Transport Instance Attributes

The *per instance* attributes to configure the plugin instance are generally passed in to the plugin constructor. These are defined by the transport plugin writer, and can be used to:

- customize the behavior of an instance of a transport plugin, including the send and the receiver buffer sizes, the maximum message size, various transport level classes of service (CoS), and so on.

- specify the resource values, network interfaces to use, various transport level policies, and so on.

RTI Connext requires that every transport plugin instance must specify the **NDDS_Transport_Property_t::message_size_max** (p. 1525) and **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).

It is up to the transport plugin developer to make these available for configuration to transport plugin user.

Note that it is important that the instance attributes are "compatible" between the sending side and the receiving side of communicating applications using different instances of a transport plugin class. For example, if one side is configured to send messages larger than can be received by the other side, then communications via the plugin may fail.

5.28.7 Transport Network Address

The address bits not used by the transport plugin for its internal addressing constitute its network address bits.

In order for RTI Connext to properly route the messages, each unicast interface in the RTI Connext *domain* must have a unique address. RTI Connext allows the user to specify the value of the network address when installing a transport plugin via the **NDDSTransportSupport::register_transport()** (p. 1560) API.

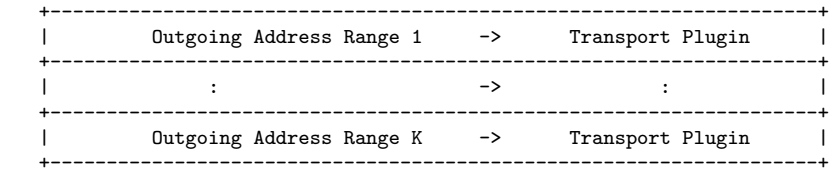
The network address for a transport plugin should be chosen such that the resulting fully qualified 128-bit address will be unique in the RTI Connext domain. Thus, if two instances of a transport plugin are registered with a **DDSDomainParticipant** (p. 1139), they will be at different network addresses in order for their unicast interfaces to have unique fully qualified 128-bit addresses. It is also possible to create multiple transports with the same network address, as it can be useful for certain use cases; note that this will require special entity configuration for most transports to avoid clashes in resource use (e.g. sockets for UDPv4 transport).

5.28.8 Transport Send Route

By default, a transport plugin is configured to send outgoing messages destined to addresses in the network address range at which the plugin was registered.

RTI Connext allows the user to configure the routing of outgoing messages via the **NDDSTransportSupport::add_send_route()** (p. 1562) API, so that a

transport plugin will be used to send messages only to certain ranges of destination addresses. The method can be called multiple times for a transport plugin, with different address ranges.

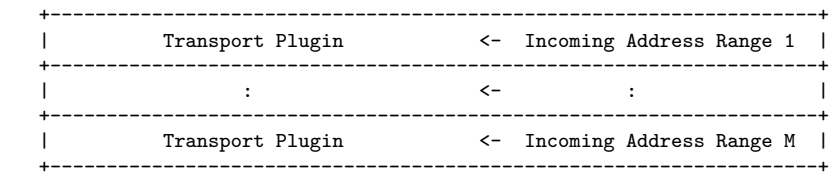


The user can set up a routing table to restrict the use of a transport plugin to send messages to selected addresses ranges.

5.28.9 Transport Receive Route

By default, a transport plugin is configured to receive incoming messages destined to addresses in the network address range at which the plugin was registered.

RTI Connext allows the user to configure the routing of incoming messages via the **NDDSTransportSupport::add_receive_route()** (p. 1563) API, so that a transport plugin will be used to receive messages only on certain ranges of addresses. The method can be called multiple times for a transport plugin, with different address ranges.



The user can set up a routing table to restrict the use of a transport plugin to receive messages from selected ranges. For example, the user may restrict a transport plugin to

- receive messages from a certain multicast address range.

- receive messages only on certain unicast interfaces (when multiple unicast interfaces are available on the transport plugin).

5.29 Using Transport Plugins

Configuring transports used by RTI Connext.

Classes

- class **NDDSTransportSupport**

  <<**interface**>> (p. 199) *The utility class used to configure RTI Connext pluggable transports.*

Typedefs

- typedef NDDS_TRANSPORT_HANDLE_TYPE_NATIVE **NDDS_Transport_Handle_t**

  *An opaque type representing the handle to a transport plugin registered with a **DDSDomainParticipant** (p. 1139).*

- typedef NDDS_Transport_Plugin *(* **NDDS_Transport_create_plugin**)(**NDDS_Transport_Address_t** *default_network_address_out, const struct **DDS_PropertyQosPolicy** *property_in)

  *Function prototype for creating plugin through **DDS_PropertyQosPolicy** (p. 834).*

Functions

- **DDS_Boolean NDDS_Transport_Handle_is_nil** (const **NDDS_Transport_Handle_t** *self)

  Is the given transport handle the NIL transport handle?

Variables

- const **NDDS_Transport_Handle_t NDDS_TRANSPORT_HANDLE_NIL**

  The NIL transport handle.

5.29.1 Detailed Description

Configuring transports used by RTI Connext.

There is more than one way to install a transport plugin for use with RTI Connext:

- If it is a builtin transport plugin, by specifying a bitmask in **DDS_TransportBuiltinQosPolicy** (p. 969) (see **Built-in Transport Plugins** (p. 136))
- For all other non-builtin transport plugins, by dynamically loading the plugin through **PROPERTY** (p. 436) QoS policy settings of **DDSDomainParticipant** (p. 1139) (on UNIX, Solaris and Windows systems only) (see **Loading Transport Plugins through Property QoS Policy of Domain Participant** (p. 131))
- By explicitly creating a transport plugin and registering the plugin with a **DDSDomainParticipant** (p. 1139) through **NDDSTransportSupport::register_transport** (p. 1560) (for both builtin and non-builtin plugins)

In the first two cases, the lifecycle of the transport plugin is automatically managed by RTI Connext. In the last case, user is responsible for deleting the transport plugin after the **DDSDomainParticipant** (p. 1139) is deleted. See **Transport Lifecycle** (p. 125) for details.

5.29.2 Loading Transport Plugins through Property QoS Policy of Domain Participant

On UNIX, Solaris and Windows operating systems, a non-builtin transport plugin written in C/C++ and built as a dynamic-link library (*.dll/*.so) can be loaded by RTI Connext through the **PROPERTY** (p. 436) QoS policy settings of the **DDSDomainParticipant** (p. 1139). The dynamic-link library (and all the dependent libraries) need to be in the path during runtime (in **LD_LIBRARY_PATH** environment variable on Linux/Solaris systems,and in **PATH** environment variable for Windows systems).

To allow dynamic loading of the transport plugin, the transport plugin must implement the RTI Connext abstract transport API and must provide a function with the signature **NDDS_Transport_create_plugin** (p. 132) that can be called by RTI Connext to create an instance of the transport plugin. The name of the dynamic library that contains the transport plugin implementation, the name of the function and properties that can be used to create the plugin, and the aliases and network address that are used to register the plugin can all be specified through the **PROPERTY** (p. 436) QoS policy of the **DDSDomainParticipant** (p. 1139).

The following table lists the property names that are used to load the transport plugins dynamically:

A transport plugin is dynamically created and registered to the **DDSDomainParticipant** (p. 1139) by RTI Connext when:

- the **DDSDomainParticipant** (p. 1139) is enabled,
- the first DataWriter/DataReader is created, or
- you lookup a builtin DataReader (**DDSSubscriber::lookup_datareader** (p. 1404)),

whichever happens first.

Any changes to the transport plugin related properties in **PROPERTY** (p. 436) QoS policy after the transport plugin has been registered with the **DDSDomainParticipant** (p. 1139) will have no effect.

See also:

Transport Use Cases (p. 181)

5.29.3 Typedef Documentation

5.29.3.1 typedef NDDS_TRANSPORT_HANDLE_TYPE_NATIVE NDDS_Transport_Handle_t

An opaque type representing the handle to a transport plugin registered with a **DDSDomainParticipant** (p. 1139).

A transport handle represents the association between a **DDSDomainParticipant** (p. 1139) and a transport plugin.

5.29.3.2 typedef NDDS_Transport_Plugin*(* NDDS_Transport_create_plugin)(NDDS_Transport_Address_t *default_network_address_out, const struct DDS_PropertyQosPolicy *property_in)

Function prototype for creating plugin through **DDS_PropertyQosPolicy** (p. 834).

By specifying some predefined property names in **DDS_PropertyQosPolicy** (p. 834), RTI Connext can call a function from a dynamic library to create a

transport plugin and register the returned plugin with a **DDSDomainParticipant** (p. 1139).

This is the function prototype of the function as specified in "<TRANSPORT_PREFIX>.create_function" of **DDS_PropertyQosPolicy** (p. 834) QoS policy that will be called by RTI Connext to create the transport plugin. See **Loading Transport Plugins through Property QoS Policy of Domain Participant** (p. 131) for details.

Parameters:

network_address_out *<<**out**>>* *(p. 200)* Optional output parameter. If the network address is not specified in "<TRANSPORT_PREFIX>.network_address" in **DDS_PropertyQosPolicy** (p. 834), this is the default network address that is used to register the returned transport plugin using **NDDSTransportSupport::register_transport** (p. 1560). This parameter will never be null. The default value is a zeroed-out network address.

property_in *<<**in**>>* *(p. 200)* **property_in** contains all the name-value pair properties that matches the format "<TRANSPORT_PREFIX>.<property_name>" in **DDS_PropertyQosPolicy** (p. 834) that can be used to create the transport plugin. Only <property_name> is passed in - the plugin prefix will be stripped out in the property name. Note: predefined <TRANSPORT_PREFIX> properties "library", "create_function", "aliases" and "network_address" will not be passed to this function. This parameter will never be null.

Returns:

Upon success, a valid non-NIL transport plugin. NIL upon failure.

5.29.4 Function Documentation

5.29.4.1 DDS_Boolean NDDS_Transport_Handle_is_nil (const NDDS_Transport_Handle_t * *self*)

Is the given transport handle the NIL transport handle?

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given transport handle is equal to **NDDS_TRANSPORT_HANDLE_NIL** (p. 134) or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

5.29.5 Variable Documentation

5.29.5.1 const NDDS_Transport_Handle_t NDDS_TRANSPORT_HANDLE_NIL

The NIL transport handle.

Property Name	Description	Required?
dds.transport.load_plugins	Comma-separated strings indicating the prefix names of all plugins that will be loaded by RTI Connext. Up to 8 plugins may be specified. For example, "dds.transport.WAN.wan1, dds.transport.DTLS.dtls1", In the following examples, <TRANSPORT_PREFIX> is used to indicate one element of this string that is used as a prefix in the property names for all the settings that are related to the plugin. <TRANSPORT_PREFIX> must begin with "dds.transport." (such as "dds.transport.mytransport").	**YES**
<TRANSPORT_PREFIX>.library	Should be set to the name of the dynamic library (*.so for Unix/Solaris, and *.dll for Windows) that contains the transport plugin implementation. This library (and all the other dependent dynamic libraries) needs to be in the path during run time for used by RTI Connext (in the **LD_LIBRARY_PATH** environment variable on UNIX/Solaris systems, in **PATH** for Windows systems).	**YES**
<TRANSPORT_PREFIX>.create_function	Should be set to the name of the function with the prototype of **NDDS_Transport_create_plugin** (p. 132) that can be called by RTI Connext to create an instance of the plugin. The resulting transport plugin will then be registered by	**YES**

5.30 Built-in Transport Plugins

Transport plugins delivered with RTI Connext.

Modules

- **Shared Memory Transport**

  *Built-in transport plug-in for inter-process communications using shared memory (**NDDS_TRANSPORT_CLASSID_SHMEM** (p. 261)).*

- **UDPv4 Transport**

  *Built-in transport plug-in using UDP/IPv4 (**NDDS_TRANSPORT_CLASSID_UDPv4** (p. 268)).*

- **UDPv6 Transport**

  *Built-in transport plug-in using UDP/IPv6 (**NDDS_TRANSPORT_CLASSID_UDPv6** (p. 278)).*

5.30.1 Detailed Description

Transport plugins delivered with RTI Connext.

The **TRANSPORT_BUILTIN** (p. 396) specifies the collection of transport plugins that can be automatically configured and managed by RTI Connext as a convenience to the user.

These transport plugins can simply be turned "on" or "off" by a specifying a bitmask in **DDS_TransportBuiltinQosPolicy** (p. 969), thus bypassing the steps for setting up a transport plugin. RTI Connext preconfigures the transport plugin properties, the network address, and the aliases to "factory defined" values.

If a builtin transport plugin is turned "on" in **DDS_TransportBuiltinQosPolicy** (p. 969), the plugin is implicitly created and registered to the corresponding **DDSDomainParticipant** (p. 1139) by RTI Connext when:

- the **DDSDomainParticipant** (p. 1139) is enabled,
- the first DataWriter/DataReader is created, or
- you lookup a builtin DataReader (**DDSSubscriber::lookup_datareader** (p. 1404)),

whichever happens first.

Each builtin transport contains its own set of properties. For example, the ::**UDPv4 Transport** (p. 265) allows the application to specify whether or not multicast is supported, the maximum size of the message, and provides a mechanism for the application to filter out network interfaces.

The builtin transport plugin properties can be changed by the method **NDDSTransportSupport::set_builtin_transport_property()** (p. 1564) or by using the **PROPERTY** (p. 436) QoS policy associated with the **DDS-DomainParticipant** (p. 1139). Builtin transport plugin properties specified in **DDS_PropertyQosPolicy** (p. 834) always overwrite the ones specified through **NDDSTransportSupport::set_builtin_transport_property()** (p. 1564). Refer to the specific builtin transport for the list of property names that can be specified through **PROPERTY** (p. 436) QoS policy.

Any changes to the builtin transport properties after the builtin transports have been registered with will have no effect.

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564) **DDS_PropertyQosPolicy** (p. 834)

The built-in transport plugins can also be instantiated and registered by the user, following the steps for **Registering a transport with a participant** (p. 183). This is useful when the application needs different values for the network addresses.

5.31 Configuration Utilities

Utility API's independent of the DDS standard.

Classes

- class **NDDSConfigVersion**
  <<**interface**>> (p. 199) *The version of an RTI Connext distribution.*
- class **NDDSConfigLoggerDevice**
  <<**interface**>> (p. 199) *Logging device interface. Use for user-defined logging devices.*
- class **NDDSConfigLogger**
  <<**interface**>> (p. 199) *The singleton type used to configure RTI Connext logging.*
- struct **NDDS_Config_LibraryVersion_t**
  The version of a single library shipped as part of an RTI Connext distribution.
- struct **NDDS_Config_LogMessage**
  Log message.

Enumerations

- enum **NDDS_Config_LogVerbosity** {
  NDDS_CONFIG_LOG_VERBOSITY_SILENT,
  NDDS_CONFIG_LOG_VERBOSITY_ERROR,
  NDDS_CONFIG_LOG_VERBOSITY_WARNING,
  NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL,
  NDDS_CONFIG_LOG_VERBOSITY_STATUS_REMOTE,
  NDDS_CONFIG_LOG_VERBOSITY_STATUS_ALL }
  The verbosities at which RTI Connext diagnostic information is logged.
- enum **NDDS_Config_LogLevel** {
  NDDS_CONFIG_LOG_LEVEL_ERROR,
  NDDS_CONFIG_LOG_LEVEL_WARNING,
  NDDS_CONFIG_LOG_LEVEL_STATUS_LOCAL,
  NDDS_CONFIG_LOG_LEVEL_STATUS_REMOTE,
  NDDS_CONFIG_LOG_LEVEL_DEBUG }

Level category assigned to RTI Connext log messages returned to an output device.

ˆ enum **NDDS_Config_LogCategory** {
NDDS_CONFIG_LOG_CATEGORY_PLATFORM,
NDDS_CONFIG_LOG_CATEGORY_COMMUNICATION,
NDDS_CONFIG_LOG_CATEGORY_DATABASE,
NDDS_CONFIG_LOG_CATEGORY_ENTITIES,
NDDS_CONFIG_LOG_CATEGORY_API }

Categories of logged messages.

ˆ enum **NDDS_Config_LogPrintFormat** { ,
NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT,
NDDS_CONFIG_LOG_PRINT_FORMAT_TIMESTAMPED,
NDDS_CONFIG_LOG_PRINT_FORMAT_VERBOSE,
NDDS_CONFIG_LOG_PRINT_FORMAT_VERBOSE_-TIMESTAMPED,
NDDS_CONFIG_LOG_PRINT_FORMAT_DEBUG,
NDDS_CONFIG_LOG_PRINT_FORMAT_MINIMAL,
NDDS_CONFIG_LOG_PRINT_FORMAT_MAXIMAL }

The format used to output RTI Connext diagnostic information.

5.31.1 Detailed Description

Utility API's independent of the DDS standard.

5.31.2 Enumeration Type Documentation

5.31.2.1 enum NDDS_Config_LogVerbosity

The verbosities at which RTI Connext diagnostic information is logged.

Enumerator:

NDDS_CONFIG_LOG_VERBOSITY_SILENT No further output will be logged.

NDDS_CONFIG_LOG_VERBOSITY_ERROR Only error messages will be logged.

An error indicates something wrong in the functioning of RTI Connext. The most common cause of errors is incorrect configuration.

NDDS_CONFIG_LOG_VERBOSITY_WARNING Both error and warning messages will be logged.

A warning indicates that RTI Connext is taking an action that may or may not be what you intended. Some configuration information is also logged at this verbosity to aid in debugging.

NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL Errors, warnings, and verbose information about the lifecycles of local RTI Connext objects will be logged.

NDDS_CONFIG_LOG_VERBOSITY_STATUS_REMOTE Errors, warnings, and verbose information about the lifecycles of remote RTI Connext objects will be logged.

NDDS_CONFIG_LOG_VERBOSITY_STATUS_ALL Errors, warnings, verbose information about the lifecycles of local and remote RTI Connext objects, and periodic information about RTI Connext threads will be logged.

5.31.2.2 enum NDDS_Config_LogLevel

Level category assigned to RTI Connext log messages returned to an output device.

Enumerator:

NDDS_CONFIG_LOG_LEVEL_ERROR The message describes an error.

An error indicates something wrong in the functioning of RTI Connext. The most common cause of errors is incorrect configuration.

NDDS_CONFIG_LOG_LEVEL_WARNING The message describes a warning.

A warning indicates that RTI Connext is taking an action that may or may not be what you intended. Some configuration information is also logged at this verbosity to aid in debugging.

NDDS_CONFIG_LOG_LEVEL_STATUS_LOCAL The message contains info about the lifecycles of local RTI Connext objects will be logged.

NDDS_CONFIG_LOG_LEVEL_STATUS_REMOTE The message contains info about the lifecycles of remote RTI Connext objects will be logged.

NDDS_CONFIG_LOG_LEVEL_DEBUG The message contains debug info that might be relevant to your application.

5.31.2.3 enum NDDS_Config_LogCategory

Categories of logged messages.

The DDSLogger::get_verbosity_by_category and DDSLogger::set_verbosity_by_-category can be used to specify different verbosities for different categories of messages.

Enumerator:

NDDS_CONFIG_LOG_CATEGORY_PLATFORM Log messages pertaining to the underlying platform (hardware and OS) on which RTI Connext is running are in this category.

NDDS_CONFIG_LOG_CATEGORY_COMMUNICATION Log messages pertaining to data serialization and deserialization and network traffic are in this category.

NDDS_CONFIG_LOG_CATEGORY_DATABASE Log messages pertaining to the internal database in which RTI Connext objects are stored are in this category.

NDDS_CONFIG_LOG_CATEGORY_ENTITIES Log messages pertaining to local and remote entities and to the discovery process are in this category.

NDDS_CONFIG_LOG_CATEGORY_API Log messages pertaining to the API layer of RTI Connext (such as method argument validation) are in this category.

5.31.2.4 enum NDDS_Config_LogPrintFormat

The format used to output RTI Connext diagnostic information.

Enumerator:

NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT Print message, method name, and activity context (default).

NDDS_CONFIG_LOG_PRINT_FORMAT_TIMESTAMPED Print message, method name, activity context, and timestamp.

NDDS_CONFIG_LOG_PRINT_FORMAT_VERBOSE Print message with all available context information (includes thread identifier, activity context).

NDDS_CONFIG_LOG_PRINT_FORMAT_VERBOSE_TIMESTAMPED Print message with all available context information, and timestamp.

NDDS_CONFIG_LOG_PRINT_FORMAT_DEBUG Print a set of field that may be useful for internal debug.

NDDS_CONFIG_LOG_PRINT_FORMAT_MINIMAL Print only message number and method name.

NDDS_CONFIG_LOG_PRINT_FORMAT_MAXIMAL Print all available fields.

5.32 Unsupported Utilities

Unsupported APIs used by examples in the RTI Connext distribution as well as in rtiddsgen-generated examples.

Classes

- class **NDDSUtility**

  Unsupported utility APIs.

5.32.1 Detailed Description

Unsupported APIs used by examples in the RTI Connext distribution as well as in rtiddsgen-generated examples.

5.33 Durability and Persistence

APIs related to RTI Connext Durability and Persistence. RTI Connext offers the following mechanisms for achieving durability and persistence:

- **Durable Writer History** (p. 144)
- **Durable Reader State** (p. 144)
- **Data Durability** (p. 145)

To use any of these features, you need a relational database, which is not included with RTI Connext. Supported databases are listed in the `Release Notes`.

These three features can be used separately or in combination.

5.33.1 Durable Writer History

This feature allows a **DDSDataWriter** (p. 1113) to locally persist its local history cache so that it can survive shutdowns, crashes and restarts. When an application restarts, each **DDSDataWriter** (p. 1113) that has been configured to have durable writer history automatically loads all the data in its history cache from disk and can carry on sending data as if it had never stopped executing. To the rest of the system, it will appear as if the **DDSDataWriter** (p. 1113) had been temporarily disconnected from the network and then reappeared.

See also:

Configuring Durable Writer History (p. 146)

5.33.2 Durable Reader State

This feature allows a **DDSDataReader** (p. 1087) to locally persists its state and remember the data it has already received. When an application restarts, each **DDSDataReader** (p. 1087) that has been configured to have durable reader state automatically loads its state from disk and can carry on receiving data as if it had never stopped executing. Data that had already been received by the **DDSDataReader** (p. 1087) before the restart will be suppressed so it is not sent over the network.

5.33.3 Data Durability

This feature is a full implementation of the OMG DDS Persistence Profile. The **DURABILITY** (p. 348) QoS lets an application configure a **DDSDataWriter** (p. 1113) such that the information written by the **DDSDataWriter** (p. 1113) survives beyond the lifetime of the **DDSDataWriter** (p. 1113). In this manner, a late-joining **DDSDataReader** (p. 1087) can subscribe and receive the information even after the **DDSDataWriter** (p. 1113) application is no longer executing. To use this feature, you need RTI Persistence Service – an optional product that can be purchased separately.

5.33.4 Durability and Persistence Based on Virtual GUID

Every modification to the global dataspace made by a **DDSDataWriter** (p. 1113) is identified by a pair (virtual GUID, sequence number).

- The virtual GUID (Global Unique Identifier) is a 16-byte character identifier associated with a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087); it is used to uniquely identify this entity in the global data space.
- The sequence number is a 64-bit identifier that identifies changes published by a specific **DDSDataWriter** (p. 1113).

Several **DDSDataWriter** (p. 1113) entities can be configured with the same virtual GUID. If each of these **DDSDataWriter** (p. 1113) entities publishes a sample with sequence number '0', the sample will only be received once by the **DDSDataReader** (p. 1087) entities subscribing to the content published by the **DDSDataWriter** (p. 1113) entities.

RTI Connext also uses the virtual GUID (Global Unique Identifier) to associate a persisted state (state in permanent storage) to the corresponding DDS entity.

For example, the history of a **DDSDataWriter** (p. 1113) will be persisted in a database table with a name generated from the virtual GUID of the **DDSDataWriter** (p. 1113). If the **DDSDataWriter** (p. 1113) is restarted, it must have associated the same virtual GUID to restore its previous history.

Likewise, the state of a **DDSDataReader** (p. 1087) will be persisted in a database table whose name is generated from the **DDSDataReader** (p. 1087) virtual GUID

A **DDSDataWriter** (p. 1113)'s virtual GUID can be configured using **DDS_DataWriterProtocolQosPolicy::virtual_guid** (p. 536). Similarly, a **DDSDataReader** (p. 1087)'s virtual GUID can be configured using **DDS_DataReaderProtocolQosPolicy::virtual_guid** (p. 502).

The **DDS_PublicationBuiltinTopicData** (p. 839) and **DDS_SubscriptionBuiltinTopicData** (p. 936) structures include the virtual

GUID associated with the discovered publication or subscription.

Refer to the `User's Manual` for additional use cases.

See also:

DDS_DataWriterProtocolQosPolicy::virtual_guid (p. 536) **DDS_DataReaderProtocolQosPolicy::virtual_guid** (p. 502).

5.33.5 Configuring Durable Writer History

To configure a **DDSDataWriter** (p. 1113) to have durable writer history, use the **PROPERTY** (p. 436) QoS policy associated with the **DDSDataWriter** (p. 1113) or the **DDSDomainParticipant** (p. 1139).

Properties defined for the **DDSDomainParticipant** (p. 1139) will be applied to all the **DDSDataWriter** (p. 1113) objects belonging to the **DDSDomainParticipant** (p. 1139), unless the property is overwritten by the **DDSDataWriter** (p. 1113).

See also:

DDS_PropertyQosPolicy (p. 834)

The following table lists the supported durable writer history properties.

5.33.6 Configuring Durable Reader State

To configure a **DDSDataReader** (p. 1087) with durable reader state, use the **PROPERTY** (p. 436) QoS policy associated with the **DDSDataReader** (p. 1087) or **DDSDomainParticipant** (p. 1139).

A property defined in the **DDSDomainParticipant** (p. 1139) will be applicable to all the **DDSDataReader** (p. 1087) belonging to the **DDSDomainParticipant** (p. 1139) unless it is overwritten by the **DDSDataReader** (p. 1087).

See also:

DDS_PropertyQosPolicy (p. 834)

The following table lists the supported durable reader state properties.

5.33.7 Configuring Data Durability

RTI Connext implements **DDS_TRANSIENT_DURABILITY_QOS** (p. 349) and **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) durability using RTI Persistence Service, available for purchase as a separate RTI product.

For more information on RTI Persistence Service, refer to the `User's Manual`, or the RTI Persistence Service API Reference HTML documentation.

See also:

DURABILITY (p. 348)

Property	Description
dds.data_writer.history.plugin_name	Must be set to "dds.data_writer.history.odbc_plugin.builtin" to enable durable writer history in the DataWriter. This property is required.
dds.data_writer.history.odbc_plugin.dsn	The ODBC DSN (Data Source Name) associated with the database where the writer history must be persisted. This property is required.
dds.data_writer.history.odbc_plugin.driver	This property tells RTI Connext which ODBC driver to load. If the property is not specified, RTI Connext will try to use the standard ODBC driver manager library: UnixOdbc (odbc32.dll) on UNIX/Linux systems; the Windows ODBC driver manager (libodbc.so) on Windows systems).
dds.data_writer.history.odbc_plugin.username	Configures the username used to connect to the database. This property is not used if it is unspecified. There is no default value.
dds.data_writer.history.odbc_plugin.password	Configures the password used to connect to the database. This property is not used if it is unspecified. There is no default value.
dds.data_writer.history.odbc_plugin.shared	If set to 1, RTI Connext creates a single connection per DSN that will be shared across DataWriters within the same Publisher. If set to 0 (the default), a **DDSDataWriter** (p. 1113) will create its own database connection. Default: 0 (false)
dds.data_writer.history.odbc_plugin.instance_cache_max_size	These properties configure the resource limits associated with the ODBC writer history caches. To minimize the number of accesses to the database, RTI Connext uses two caches, one for samples and one for instances. The initial and maximum sizes of these caches are configured using these properties. The resource limits initial_instances, max_instances, initial_samples, max_samples and max_samples_per_instance in the **DDS_ResourceLimitsQosPolicy** (p. 879) are used to configure the maximum number of samples and instances that can be stored in the relational database. Default: **DDS_ResourceLimitsQosPolicy::max_**

Property	Description
dds.data_reader.state.odbc.dsn	The ODBC DSN (Data Source Name) associated with the database where the **DDSDataReader** (p. 1087) state must be persisted. This property is required.
dds.data_reader.state.filter_-redundant_samples	To enable durable reader state, this property must be set to 1. Otherwise, the reader state will not be kept and/or persisted. When the reader state is not maintained, RTI Connext does not filter duplicate samples that may be coming from the same virtual writer. By default, this property is set to 1.
dds.data_reader.state.odbc.driver	This property is used to indicate which ODBC driver to load. If the property is not specified, RTI Connext will try to use the standard ODBC driver manager library: UnixOdbc (odbc32.dll) on UNIX/Linux systems; the Windows ODBC driver manager (libodbc.so) on Windows systems).
dds.data_-reader.state.odbc.username	This property configures the username used to connect to the database. This property is not used if it is unspecified. There is no default value.
dds.data_reader.state.odbc.password	This property configures the password used to connect to the database. This property is not used if it is unspecified. There is no default value.
dds.data_reader.state.restore	This property indicates if the persisted **DDSDataReader** (p. 1087) state must be restored or not once the **DDSDataReader** (p. 1087) is restarted. If this property is 0, the previous state will be deleted from the database. If it is 1, the **DDSDataReader** (p. 1087) will restore its previous state from the database content. Default: 1
dds.data_reader.state.checkpoint_-frequency	This property controls how often the reader state is stored in the database. A value of N means to store the state once every N samples. A high frequency will provide better performance. However, if the reader is restarted it may receive some duplicate samples. These samples will be filtered by the middleware and they will not be propagated to the application.

5.34 System Properties

System Properties. RTI Connext uses the **DDS_PropertyQosPolicy** (p. 834) of a DomainParticipant to maintain a set of properties that provide system information such as hostname.

Unless the default **DDS_DomainParticipantQos** (p. 588) value is overwritten, the system properties are automatically set in the **DDS_DomainParticipantQos** (p. 588) obtained by calling the method **DDSDomainParticipantFactory::get_default_participant_qos** (p. 1224) or using the constant **DDS_PARTICIPANT_QOS_DEFAULT** (p. 35).

System properties are also automatically set in the **DDS_DomainParticipantQos** (p. 588) loaded from an XML QoS profile unless you disable property inheritance using the attribute **inherit** in the XML tag <**property**>.

By default, the system properties are propagated to other DomainParticipants in the system and can be accessed through **DDS_ParticipantBuiltinTopicData::property** (p. 817).

You can disable the propagation of individual properties by setting the flag **DDS_Property_t::propagate** (p. 833) to **DDS_BOOLEAN_FALSE** (p. 299) or by removing the property using the method DDS_PropertyQosPolicyHelper::remove_property.

The number of system properties set on the **DDS_DomainParticipantQos** (p. 588) is platform specific.

5.34.1 System Properties List

The following table lists the supported system properties.

Property Name	Description
dds.sys_info.hostname	Hostname
dds.sys_info.process_id	Process ID

5.34.2 System Resource Consideration

System properties are affected by the resource limits **DDS_DomainParticipantResourceLimitsQosPolicy::participant_property_list_max_length** (p. 610) and **DDS_DomainParticipantResourceLimitsQosPolicy::participant_property_string_max_length** (p. 610).

5.35 Configuring QoS Profiles with XML

APIs related to XML QoS Profiles.

5.35.1 Loading QoS Profiles from XML Resources

A 'QoS profile' is a group of QoS settings, specified in XML format. By using QoS profiles, you can change QoS settings without recompiling the application.

The Qos profiles are loaded the first time any of the following operations are called:

- **DDSDomainParticipantFactory::create_participant** (p. 1233)
- **DDSDomainParticipantFactory::create_participant_with_profile** (p. 1235)
- **DDSDomainParticipantFactory::set_default_participant_qos_with_profile** (p. 1223)
- **DDSDomainParticipantFactory::get_default_participant_qos** (p. 1224)
- **DDSDomainParticipantFactory::set_default_library** (p. 1225)
- **DDSDomainParticipantFactory::set_default_profile** (p. 1225)
- **DDSDomainParticipantFactory::get_participant_qos_from_profile** (p. 1227)
- **DDSDomainParticipantFactory::get_topic_qos_from_profile** (p. 1231)
- **DDSDomainParticipantFactory::get_topic_qos_from_profile_w_topic_name** (p. 1232)
- **DDSDomainParticipantFactory::get_publisher_qos_from_profile** (p. 1228)
- **DDSDomainParticipantFactory::get_subscriber_qos_from_profile** (p. 1228)
- **DDSDomainParticipantFactory::get_datawriter_qos_from_profile** (p. 1229)
- **DDSDomainParticipantFactory::get_datawriter_qos_from_profile_w_topic_name** (p. 1229)
- **DDSDomainParticipantFactory::get_datareader_qos_from_profile** (p. 1230)

- **DDSDomainParticipantFactory::get_datareader_qos_from_profile_w_topic_name** (p. 1231)
- **DDSDomainParticipantFactory::get_qos_profile_libraries** (p. 1233)
- **DDSDomainParticipantFactory::get_qos_profiles** (p. 1233)
- **DDSDomainParticipantFactory::load_profiles** (p. 1238)

The QoS profiles are reloaded replacing previously loaded profiles when the following operations are called:

- **DDSDomainParticipantFactory::set_qos** (p. 1237)
- **DDSDomainParticipantFactory::reload_profiles** (p. 1238)

The **DDSDomainParticipantFactory::unload_profiles()** (p. 1239) operation will free the resources associated with the XML QoS profiles.

There are five ways to configure the XML resources (listed by load order):

- The file NDDS_QOS_PROFILES.xml in $NDDSHOME/resource/qos_profiles_5.0.0/xml is loaded if it exists and **DDS_ProfileQosPolicy::ignore_resource_profile** (p. 832) in **DDS_ProfileQosPolicy** (p. 830) is set to **DDS_BOOLEAN_FALSE** (p. 299) (first to be loaded). An example file, `NDDS_QOS_PROFILES.example.xml`, is available for reference.

- The URL groups separated by semicolons referenced by the enviroment variable NDDS_QOS_PROFILES are loaded if they exist and **DDS_ProfileQosPolicy::ignore_environment_profile** (p. 831) in **DDS_ProfileQosPolicy** (p. 830) is set to **DDS_BOOLEAN_FALSE** (p. 299).

- The file USER_QOS_PROFILES.xml in the working directory will be loaded if it exists and **DDS_ProfileQosPolicy::ignore_user_profile** (p. 831) in **DDS_ProfileQosPolicy** (p. 830) is set to **DDS_BOOLEAN_FALSE** (p. 299).

- The URL groups referenced by **DDS_ProfileQosPolicy::url_profile** (p. 831) in **DDS_ProfileQosPolicy** (p. 830) will be loaded if specified.

- The sequence of XML strings referenced by **DDS_ProfileQosPolicy::string_profile** (p. 831) will be loaded if specified (last to be loaded).

The above methods can be combined together.

5.35.2 URL

The location of the XML resources (only files and strings are supported) is specified using a URL (Uniform Resource Locator) format. For example:

File Specification: `file:///usr/local/default_dds.xml`

String Specification: str://"<dds><qos_library> . . . lt;/qos_-library>&lt/dds>"

If the URL schema name is omitted, RTI Connext will assume a file name. For example:

File Specification: /usr/local/default_dds.xml

5.35.2.1 URL groups

To provide redundancy and fault tolerance, you can specify multiple locations for a single XML document via URL groups. The syntax of a URL group is as follows:

[URL1 | URL2 | URL2 | . . .| URLn]

For example:

[`file:///usr/local/default_dds.xml` | `file:///usr/local/alternative_-default_dds.xml`]

Only one of the elements in the group will be loaded by RTI Connext, starting from the left.

Brackets are not required for groups with a single URL.

5.35.2.2 NDDS_QOS_PROFILES environment variable

The environment variable NDDS_QOS_PROFILES contains a list of URL groups separated by ';'

The URL groups referenced by the environment variable are loaded if they exist and **DDS_ProfileQosPolicy::ignore_environment_profile** (p. 831) is set to **DDS_BOOLEAN_FALSE** (p. 299)

For more information on XML Configuration, refer to the `User's Manual`.

5.36 Publication Example

A data publication example.

5.36.1 A typical publication example

Prep

- **Create user data types using rtiddsgen** (p. 220)

Set up

- **Get the factory** (p. 156)
- **Set up participant** (p. 156)
- **Set up publisher** (p. 165)
- **Register user data type(s)** (p. 159)
- **Set up topic(s)** (p. 159)
- **Set up data writer(s)** (p. 166)

Adjust the desired quality of service (QoS)

- **Adjust QoS on entities as necessary** (p. 176)

Send data

- **Send data** (p. 167)

Tear down

- **Tear down data writer(s)** (p. 167)
- **Tear down topic(s)** (p. 160)
- **Tear down publisher** (p. 165)
- **Tear down participant** (p. 157)

5.37 Subscription Example

A data subscription example.

5.37.1 A typical subscription example

Prep

- **Create user data types using rtiddsgen** (p. 220)

Set up

- **Get the factory** (p. 156)
- **Set up participant** (p. 156)
- **Set up subscriber** (p. 168)
- **Register user data type(s)** (p. 159)
- **Set up topic(s)** (p. 159)
- **Set up data reader(s)** (p. 172)
- **Set up data reader** (p. 173) **OR Set up subscriber** (p. 168) to receive data

Adjust the desired quality of service (QoS)

- **Adjust QoS on entities as necessary** (p. 176)

Receive data

- Access received data either **via a reader** (p. 173) **OR via a subscriber** (p. 169) (possibly in a **ordered or coherent** (p. 170) manner)

Tear down

- **Tear down data reader(s)** (p. 175)
- **Tear down topic(s)** (p. 160)
- **Tear down subscriber** (p. 170)
- **Tear down participant** (p. 157)

5.38 Participant Use Cases

Working with domain partcipants. Working with domain participants.

5.38.1 Turning off auto-enable of newly created participant(s)

- **Get the factory** (p. 156)
- Change the value of the **ENTITY_FACTORY** (p. 377) for the **DDSDomainParticipantFactory** (p. 1216)

```
DDS_DomainParticipantFactoryQos factory_qos;

if (factory->get_qos(factory_qos) != DDS_RETCODE_OK) {
     printf("***Error: failed to get domain participant factory qos\n");
}

/* Change the QosPolicy to create disabled participants */
factory_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;

if (factory->set_qos(factory_qos) != DDS_RETCODE_OK) {
    printf("***Error: failed to set domain participant factory qos\n");
}
```

5.38.2 Getting the factory

- Get the **DDSDomainParticipantFactory** (p. 1216) instance:

```
DDSDomainParticipantFactory* factory = NULL;

factory = DDSDomainParticipantFactory::get_instance();

if (factory == NULL) {
   // ... error
}
```

5.38.3 Setting up a participant

- **Get the factory** (p. 156)
- Create **DDSDomainParticipant** (p. 1139):

```
DDS_DomainParticipantQos participant_qos;
   DDSDomainParticipantListener* participant_listener = NULL;
   DDS_ReturnCode_t retcode;
```

```
 // Set the initial peers. This list includes all the computers the
 // application may communicate with, along with the maximum number of
 // RTI Data Distribution Service participants that can concurrently
 // run on that computer. This list only needs to be a superset of the
 // actual list of computers and participants that will be running at
 // any time.

 const char* NDDS_DISCOVERY_INITIAL_PEERS[] = {
  "host1",
  "10.10.30.192",
  "1@localhost",
  "2@host2",
  "my://", /* all unicast addrs on transport plugins with alias "my" */
  "2@shmem://", /* shared memory */
  "FF00:ABCD::0",
  "sf://0/0/R", /* StarFabric transport plugin */
  "1@FF00:0:1234::0",
  "225.1.2.3",
  "3@225.1.0.55",
  "FAA0::0#0/0/R",
};

const DDS_Long NDDS_DISCOVERY_INITIAL_PEERS_LENGTH =
      sizeof(NDDS_DISCOVERY_INITIAL_PEERS)/sizeof(const char*);

// initialize participant_qos with default values
retcode = factory->get_default_participant_qos(participant_qos);

if (retcode != DDS_RETCODE_OK) {
    printf("***Error: failed to get default participant qos\n");
}

if (!participant_qos.discovery.initial_peers.from_array(
                        NDDS_DISCOVERY_INITIAL_PEERS,
                         NDDS_DISCOVERY_INITIAL_PEERS_LENGTH)) {
    printf("***Error: failed to set discovery.initial_peers qos\n");
}

// Create the participant
DDSDomainParticipant* participant =
    factory->create_participant(domain_id,
                                participant_qos,
                                participant_listener,
                                        DDS_STATUS_MASK_NONE);
if (participant == NULL) {
     printf("***Error: failed to create domain participant\n");
};
return participant;
```

5.38.4 Tearing down a participant

- ^ **Get the factory** (p. 156)

- ^ Delete **DDSDomainParticipant** (p. 1139):

```
DDS_ReturnCode_t retcode;

retcode = factory->delete_participant(participant);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.39 Topic Use Cases

Working with topics.

5.39.1 Registering a user data type

- **Set up participant** (p. 156)

- Register user data type of type `T` under the name "My_Type"

```
const char* type_name = "My_Type";
DDS_ReturnCode_t retcode;

retcode = FooTypeSupport::register_type(participant, type_name);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
```

5.39.2 Setting up a topic

- **Set up participant** (p. 156)

- **Ensure user data type is registered** (p. 159)

- Create a **DDSTopic** (p. 1419) under the name "my_topic"

```
const char* topic_name = "my_topic";
const char* type_type = "My_Type";   // user data type
DDS_TopicQos topic_qos;
DDS_ReturnCode_t retcode;

// MyTopicListener is user defined and
// extends DDSTopicListener
DDSTopicListener* topic_listener = new MyTopicListener(); // or = NULL

retcode = participant->get_default_topic_qos(topic_qos);

if (retcode != DDS_RETCODE_OK) {
    // ... error
}

DDSTopic* topic = participant->create_topic(topic_name, type_name,
                                            topic_qos, topic_listener,
                                            DDS_STATUS_MASK_ALL);
if (topic == NULL) {
    // ... error

};
```

5.39.3 Tearing down a topic

ˆ Delete **DDSTopic** (p. 1419):

```
DDS_ReturnCode_t retcode;

retcode = participant->delete_topic(topic);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.40 FlowController Use Cases

Working with flow controllers.

5.40.1 Creating a flow controller

- **Set up participant** (p. 156)

- Create a flow controller

```
DDS_ReturnCode_t retcode;
DDSFlowController *controller = NULL;
DDS_FlowControllerProperty_t property;

retcode = participant->get_default_flowcontroller_property(property);

if (retcode != DDS_RETCODE_OK) {
    printf("***Error: failed to get default flow controller property\n");
}

// optionally modify flow controller property values

controller = participant->create_flowcontroller(
    "my flow controller name", property);

if (controller == NULL) {
    printf("***Error: failed to create flow controller\n");
}
```

5.40.2 Flow controlling a data writer

- **Set up participant** (p. 156)

- **Create flow controller** (p. 161)

- Create an asynchronous data writer, **FooDataWriter** (p. 1475), of user data type `Foo` (p. 1443):

```
DDS_DataWriterQos writer_qos;
DDS_ReturnCode_t retcode;

// MyWriterListener is user defined and
// extends DDSDataWriterListener
MyWriterListener* writer_listener = new MyWriterListener(); // or = NULL

retcode = publisher->get_default_datawriter_qos(writer_qos);

if (retcode != DDS_RETCODE_OK) {
    // ... error
```

```
}

/* Change the writer QoS to publish asnychronously */
writer_qos.publish_mode.kind = DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS;

/* Setup to use the previously created flow controller */
writer_qos.publish_mode.flow_controller_name =
    DDS_String_dup("my flow controller name");

/* Samples queued for asynchronous write are subject to the History Qos policy */
writer_qos.history.kind = DDS_KEEP_ALL_HISTORY_QOS;

FooDataWriter* writer = publisher->create_datawriter(topic,
                                                     writer_qos,
                                                     writer_listener,
                          DDS_STATUS_MASK_ALL);
if (writer == NULL) {
    // ... error
};

/* Send data asynchronously... */

/* Wait for asynchronous send completes, if desired */
retcode = writer->wait_for_asynchronous_publishing(timout);

if (retcode != DDS_RETCODE_OK) {
    printf("***Error: failed to wait for asynchronous publishing\n");
}
```

5.40.3 Using the built-in flow controllers

RTI Connext provides several built-in flow controllers.

The **DDS_DEFAULT_FLOW_CONTROLLER_NAME** (p. 91) built-in flow controller provides the basic asynchronous writer behavior. When calling **FooDataWriter::write** (p. 1484), the call signals the **DDSPublisher** (p. 1346) asynchronous publishing thread (**DDS_PublisherQos::asynchronous_publisher** (p. 852)) to send the actual data. As with any **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) **DDSDataWriter** (p. 1113), the **FooDataWriter::write** (p. 1484) call returns immediately afterwards. The data is sent immediately in the context of the **DDSPublisher** (p. 1346) asynchronous publishing thread.

When using the **DDS_FIXED_RATE_FLOW_CONTROLLER_NAME** (p. 92) flow controller, data is also sent in the context of the **DDSPublisher** (p. 1346) asynchronous publishing thread, but at a regular fixed interval. The thread accumulates samples from different **DDSDataWriter** (p. 1113) instances and generates data on the wire only once per **DDS_FlowControllerTokenBucketProperty_t::period** (p. 753).

In contrast, the **DDS_ON_DEMAND_FLOW_CONTROLLER_NAME** (p. 93) flow controller permits flow only when **DDSFlowController::trigger_**

flow (p. 1261) is called. The data is still sent in the context of the **DDSPublisher** (p. 1346) asynchronous publishing thread. The thread accumulates samples from different **DDSDataWriter** (p. 1113) instances (across any **DDSPublisher** (p. 1346)) and sends all data since the previous trigger.

The properties of the built-in **DDSFlowController** (p. 1259) instances can be adjusted.

- **Set up participant** (p. 156)
- Lookup built-in flow controller

```
DDSFlowController *controller = NULL;

controller = participant->lookup_flowcontroller(
    DDS_DEFAULT_FLOW_CONTROLLER_NAME);

/* This should never happen, built-in flow controllers are always created */
if (controller == NULL) {
    printf("***Error: failed to lookup flow controller\n");
}
```

- Change property of built-in flow controller, if desired

```
DDS_ReturnCode_t retcode;
DDS_FlowControllerProperty_t property;

/* Get the property of the flow controller */
retcode = controller->get_property(property);

if (retcode != DDS_RETCODE_OK) {
    printf("***Error: failed to get flow controller property\n");
}

/* Change the property value as desired */
property.token_bucket.period.sec = 2;
property.token_bucket.period.nanosec = 0;

/* Update the flow controller property */
retcode = controller->set_property(property);

if (retcode != DDS_RETCODE_OK) {
    printf("***Error: failed to set flow controller property\n");
}
```

- **Create a data writer using the correct flow controller name** (p. 161)

5.40.4 Shaping the network traffic for a particular transport

- **Set up participant** (p. 156)

- **Create the transports** (p. 181)
- **Create a separate flow controller for each transport** (p. 161)
- Configure **DDSDataWriter** (p. 1113) instances to only use a single transport
- **Associate all data writers using the same transport to the corresponding flow controller** (p. 161)
- For each transport, the corresponding flow controller limits the network traffic based on the token bucket properties

5.40.5 Coalescing multiple samples in a single network packet

- **Set up participant** (p. 156)
- **Create a flow controller with a desired token bucket period** (p. 161)
- **Associate the data writer with the flow controller** (p. 161)
- Multiple samples written within the specified period will be coalesced into a single network packet (provided that `tokens_added_per_period` and `bytes_per_token` permit).

5.41 Publisher Use Cases

Working with publishers.

5.41.1 Setting up a publisher

- **Set up participant** (p. 156)

- Create a **DDSPublisher** (p. 1346)

```
DDS_PublisherQos publisher_qos;

// MyPublisherListener is user defined and
// extends DDSPublisherListener
DDSPublisherListener* publisher_listener =
              =                    new MyPublisherListener(); // or = NULL

participant->get_default_publisher_qos(publisher_qos);

DDSPublisher* publisher = participant->create_publisher(publisher_qos,
                                                        publisher_listener,
                                                       DDS_STATUS_MASK_ALL);
if (publisher == NULL) {
    // ... error

};
```

5.41.2 Tearing down a publisher

- Delete **DDSPublisher** (p. 1346):

```
DDS_ReturnCode_t retcode;

retcode = participant->delete_publisher(publisher);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.42 DataWriter Use Cases

Working with data writers.

5.42.1 Setting up a data writer

- **Set up publisher** (p. 165)
- **Set up a topic** (p. 159)
- Create a data writer, **FooDataWriter** (p. 1475), of user data type `Foo` (p. 1443):

```
DDS_DataWriterQos writer_qos;
DDS_ReturnCode_t retcode;

// MyWriterListener is user defined and
// extends DDSDataWriterListener
MyWriterListener* writer_listener = new MyWriterListener(); // or = NULL

retcode = publisher->get_default_datawriter_qos(writer_qos);

if (retcode != DDS_RETCODE_OK) {
    // ... error
}

FooDataWriter* writer = publisher->create_datawriter(topic,
                                                     writer_qos,
                                                     writer_listener,
                                                       DDS_STATUS_MASK_ALL);
if (writer == NULL) {
    // ... error

};
```

5.42.2 Managing instances

- Getting an instance "key" value of user data type `Foo` (p. 1443)

```
Foo* data = ...;     // user data

retcode = writer->get_key_value(*data, instance_handle);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

- Registering an instance of type `Foo` (p. 1443)

```
DDS_InstanceHandle_t instance_handle = DDS_HANDLE_NIL;

instance_handle = writer->register_instance(data);
```

- Unregistering an instance of type `Foo` (p. 1443)

```
retcode = writer->unregister_instance(data, instance_handle);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

- Disposing of an instance of type `Foo` (p. 1443)

```
retcode = writer->dispose(data, instance_handle);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.42.3 Sending data

- **Set up data writer** (p. 166)
- **Register instance** (p. 166)
- Write instance of type `Foo` (p. 1443)

```
Foo* data;     // user data

DDS_InstanceHandle_t instance_handle =
                       DDS_HANDLE_NIL; // or a valid registered handle

DDS_ReturnCode_t retcode;

retcode = writer->write(data, instance_handle);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.42.4 Tearing down a data writer

- Delete **DDSDataWriter** (p. 1113):

```
DDS_ReturnCode_t retcode;

retcode = publisher->delete_datawriter(writer);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.43 Subscriber Use Cases

Working with subscribers.

5.43.1 Setting up a subscriber

- **Set up participant** (p. 156)

- Create a **DDSSubscriber** (p. 1390)

```
DDS_SubscriberQos subscriber_qos;
DDS_ReturnCode_t retcode;

// MySubscriberListener is user defined and
// extends DDSSubscriberListener
DDSSubscriberListener* subscriber_listener =
                                new MySubscriberListener(); // or = NULL

retcode = participant->get_default_subscriber_qos(subscriber_qos);

if (retcode != DDS_RETCODE_OK) {
    // ... error

};

DDSSubscriber* subscriber =
                      participant->create_subscriber(subscriber_qos,
                                                     subscriber_listener,
                                                     DDS_STATUS_MASK_ALL);
if (subscriber == NULL) {
    // ... error

};
```

5.43.2 Set up subscriber to access received data

- **Set up subscriber** (p. 168)

- Set up to handle the DDS_DATA_ON_READERS_STATUS status, in one or both of the following two ways.

- **Enable DDS_DATA_ON_READERS_STATUS for the DDSSubscriberListener associated with the subscriber** (p. 177)

  - The processing to handle the status change is done in the **DDSSubscriberListener::on_data_on_readers()** (p. 1415) method of the attached listener.

- Typical processing will **access the received data** (p. 169), either in arbitrary order or in a **coherent and ordered manner** (p. 170).

ˆ **Enable DDS_DATA_ON_READERS_STATUS for the DDSStatusCondition associated with the subscriber** (p. 178)

- The processing to handle the status change is done **when the subscriber's attached status condition is triggered** (p. 179) and the DDS_DATA_ON_READERS_STATUS status on the subscriber is changed.
- Typical processing will **access the received data** (p. 169), either in an arbitrary order or in a **coherent and ordered manner** (p. 170).

5.43.3 Access received data via a subscriber

ˆ **Ensure subscriber is set up to access received data** (p. 168)

ˆ Get the list of readers that have data samples available:

```
DDSDataReaderSeq reader_seq;  // holder for list/set of readers
DDS_SampleStateMask     sample_state_mask = DDS_NOT_READ_SAMPLE_STATE;
DDS_ViewStateMask       view_state_mask = DDS_ANY_VIEW_STATE;
DDS_InstanceStateMask   instance_state_mask = DDS_ANY_INSTANCE_STATE;


DDS_ReturnCode_t retcode;

// get_datareadersX is not supported yet.
retcode = subscriber->get_datareaders(reader_seq,
                                      sample_state_mask,
                                      view_state_mask,
                                      instance_state_mask);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

ˆ Upon successfully getting the list of readers with data, process the data readers to either:

- **Read the data in each reader** (p. 174), **OR**
- **Take the data in each reader** (p. 173)

If the intent is to access the data **coherently or in order** (p. 170), the list of data readers *must* be processed in the order returned:

```
for(int i = 0; i < reader_seq.length(); ++i) {
```

```
    TDataReader* reader = reader_seq[i];

    // Take the data from reader,
    //            OR
    // Read the data from reader
}
```

- **Alternatively**, call **DDSSubscriber::notify_datareaders()** (p. 1408) to invoke the **DDSDataReaderListener** (p. 1108) for each of the data readers.

```
retcode = subscriber->notify_datareaders();

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.43.4 Access received data coherently and/or in order

To access the received data coherently and/or in an ordered manner, according to the settings of the **DDS_PresentationQosPolicy** (p. 823) attached to a **DDSSubscriber** (p. 1390):

- **Ensure subscriber is set up to access received data** (p. 168)

- Indicate that data will be accessed via the subscriber:

```
retcode = subscriber->begin_access();

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

- **Access received data via the subscriber, making sure that the data readers are processed in the order returned.** (p. 169)

- Indicate that the data access via the subscriber is done:

```
retcode = subscriber->end_access();

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.43.5 Tearing down a subscriber

- Delete **DDSSubscriber** (p. 1390):

```
DDS_ReturnCode_t retcode;

retcode = participant->delete_subscriber(subscriber);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.44 DataReader Use Cases

Working with data readers.

5.44.1 Setting up a data reader

- **Set up subscriber** (p. 168)
- **Set up a topic** (p. 159)
- Create a data reader, **FooDataReader** (p. 1444), of user data type `Foo` (p. 1443):

```
DDS_DataReaderQos reader_qos;
DDS_ReturnCode_t retcode;

// MyReaderListener is user defined and
// extends DDSDataReaderListener
DDSDataReaderListener* reader_listener =
      new MyReaderListener(); // or = NULL

retcode = subscriber->get_default_datareader_qos(reader_qos);

if (retcode != DDS_RETCODE_OK) {
    // ... error

};

FooDataReader* reader = subscriber->create_datareader(topic,
                                                      reader_qos,
                                                      reader_listener,
                                                      DDS_STATUS_MASK_ALL);
if (reader == NULL) {
    // ... error

};
```

5.44.2 Managing instances

- Given a data reader

```
FooDataReader* reader = ...;
```

- Getting an instance "key" value of user data type `Foo` (p. 1443)

```
Foo data;     // user data of type Foo
// ...
retcode = reader->get_key_value(data, instance_handle);
```

```
if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.44.3 Set up reader to access received data

- **Set up data reader** (p. 172)
- Set up to handle the DDS_DATA_AVAILABLE_STATUS status, in one or both of the following two ways.
- **Enable DDS_DATA_AVAILABLE_STATUS for the DDSDataReaderListener associated with the data reader** (p. 177)
  - The processing to handle the status change is done in the **DDSDataReaderListener::on_data_available** (p. 1110) method of the attached listener.
  - Typical processing will **access the received data** (p. 173).
- **Enable DDS_DATA_AVAILABLE_STATUS for the DDSStatusCondition associated with the data reader** (p. 178)
  - The processing to handle the status change is done **when the data reader's attached status condition is triggered** (p. 179) and the DDS_DATA_AVAILABLE_STATUS status on the data reader is changed.
  - Typical processing will **access the received data** (p. 173).

5.44.4 Access received data via a reader

- **Ensure reader is set up to access received data** (p. 173)
- Access the received data, by either:
  - **Taking the received data in the reader** (p. 173), **OR**
  - **Reading the received data in the reader** (p. 174)

5.44.5 Taking data

- **Ensure reader is set up to access received data** (p. 173)
- Take samples of user data type `T`. The samples are removed from the Service. The caller is responsible for deallocating the buffers.

```
FooSeq                data_seq; // holder for sequence of user data type Foo
DDS_SampleInfoSeq     info_seq; // holder for sequence of DDS_SampleInfo
long                  max_samples       = DDS_LENGTH_UNLIMITED;
DDS_SampleStateMask   sample_state_mask = DDS_ANY_SAMPLE_STATE;
DDS_ViewStateMask     view_state_mask = DDS_ANY_VIEW_STATE;
DDS_InstanceStateMask instance_state_mask = DDS_ANY_INSTANCE_STATE;

DDS_ReturnCode_t retcode;

retcode = reader->take(data_seq, info_seq,
                       max_samples,
                       sample_state_mask,
                       view_state_mask,
                       instance_state_mask);

if (retcode == DDS_RETCODE_NO_DATA) {
    return;
} else if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

ˆ Use the received data

```
// Use the received data samples 'data_seq' and associated information 'info_seq'
for(int i = 0; i < data_seq.length(); ++i) {
    // use... data_seq[i] ...
    // use... info_seq[i] ...
}
```

ˆ Return the data samples and the information buffers back to the middleware. *IMPORTANT*: Once this call returns, you must not retain any pointers to any part of any sample or sample info object.

```
retcode = reader->return_loan(data_seq, info_seq);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.44.6 Reading data

ˆ **Ensure reader is set up to access received data** (p. 173)

ˆ Read samples of user data type `Foo` (p. 1443). The samples are not removed from the Service. It remains responsible for deallocating the buffers.

```
FooSeq                data_seq; // holder for sequence of user data type Foo
DDS_SampleInfoSeq     info_seq; // holder for sequence of DDS_SampleInfo
long                  max_samples       = DDS_LENGTH_UNLIMITED;
DDS_SampleStateMask   sample_state_mask = DDS_NOT_READ_SAMPLE_STATE;
```

```
DDS_ViewStateMask     view_state_mask = DDS_ANY_VIEW_STATE;
DDS_InstanceStateMask instance_state_mask = DDS_ANY_INSTANCE_STATE;

DDS_ReturnCode_t retcode;

retcode = reader->read(data_seq, info_seq,
                       max_samples,
                       sample_state_mask,
                       view_state_mask,
                       instance_state_mask);

if (retcode == DDS_RETCODE_NO_DATA) {
    return;
} else if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

- Use the received data

```
// Use the received data samples 'data_seq' and associated
// information 'info_seq'
for(int i = 0; i < data_seq.length(); ++i) {
    // use... data_seq[i] ...
    // use... info_seq[i] ...
}
```

- Return the data samples and the information buffers back to the middleware

```
retcode = reader->return_loan(data_seq, info_seq);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.44.7 Tearing down a data reader

- Delete **DDSDataReader** (p. 1087):

```
DDS_ReturnCode_t retcode;

retcode = subscriber->delete_datareader(reader);

if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
}
```

5.45 Entity Use Cases

Working with entities.

5.45.1 Enabling an entity

ˆ To enable an **DDSEntity** (p. 1253)

```
if (entity->enable() != DDS_RETCODE_OK) {
    printf("***Error: failed to enable entity\n");
}
```

5.45.2 Checking if a status changed on an entity.

ˆ Given an **DDSEntity** (p. 1253) and a **DDS_StatusKind** (p. 322) to check for, get the list of statuses that have changed since the last time they were respectively cleared.

```
DDS_StatusMask status_changes_mask = entity->get_status_changes();
```

ˆ Check if `status_kind` was changed since the last time it was cleared. A plain communication status change is cleared when the status is read using the entity's `get_<plain communication status>()` method. A read communication status change is cleared when the data is taken from the middleware via a TDataReader_take() call [see **Changes in Status** (p. 319) for details].

```
if (status_changes_mask & status_kind) {
    return true;
} else {           /* ... YES, status_kind changed ... */
    return false;  /* ... NO, status_kind did NOT change ... */
}
```

5.45.3 Changing the QoS for an entity

The QoS for an entity can be specified at the entity creation time. Once an entity has been created, its QoS can be manipulated as follows.

ˆ Get an entity's QoS settings using **get_qos (abstract)** (p. 1255)

```
if (entity->get_qos(qos) != DDS_RETCODE_OK) {
    printf("***Error: failed to get qos\n");
}
```

ˆ Change the desired qos policy fields

```
/* Change the desired qos policies */
/* qos.policy.field = ... */
```

- Set the qos using **set_qos (abstract)** (p. 1254).

```
switch (entity->set_qos(qos)) {
    case DDS_RETCODE_OK: { /* success */
    } break;
    case DDS_RETCODE_IMMUTABLE_POLICY: {
        printf("***Error: tried changing a policy that can only be"
               "          set at entity creation time\n");
    } break;
    case DDS_RETCODE_INCONSISTENT_POLICY: {
        printf("***Error: tried changing a policy to a value inconsistent"
               "          with other policy settings\n");
    } break;
    default: {
        printf("***Error: some other failure\n");
    }
}
```

5.45.4 Changing the listener and enabling/disabling statuses associated with it

The listener for an entity can be specified at the entity creation time. By default the listener is *enabled* for all the statuses supported by the entity.

Once an entity has been created, its listener and/or the statuses for which it is enabled can be manipulated as follows.

- User defines entity listener methods

```
/* ... methods defined by EntityListener ... */
public class MyEntityListener implements DDSListener {
    // ... methods defined by EntityListener ...
}
```

- Get an entity's listener using **get_listener (abstract)** (p. 1256)

```
entity_listener = entity->get_listener();
```

- Enable `status_kind` for the `listener`

```
enabled_status_list |= status_kind;
```

- Disable `status_kind` for the `listener`

```
enabled_status_list &= ~status_kind;
```

- Set an entity's listener to `entity_listener` using **set_listener (abstract)** (p. 1255). Only enable the listener for the statuses specified by the `enabled_status_list`.

```
if (entity->set_listener(entity_listener, enabled_status_list)
            != DDS_RETCODE_OK) {
    printf("***Error: setting entity listener\n");
}
```

5.45.5 Enabling/Disabling statuses associated with a status condition

Upon entity creation, by default, all the statuses are *enabled* for the DDS_StatusCondition associated with the entity.

Once an entity has been created, the list of statuses for which the DDS_StatusCondition is triggered can be manipulated as follows.

- Given an `entity`, a `status_kind`, and the associated `status_condition`:

```
statuscondition = entity->get_statuscondition();
```

- Get the list of statuses enabled for the `status_condition`

```
enabled_status_list = statuscondition->get_enabled_statuses();
```

- Check if the given `status_kind` is enabled for the `status_condition`

```
if (enabled_status_list & status_kind) {
    /*... YES, status_kind is enabled ... */
} else {
    /* ... NO, status_kind is NOT enabled ... */
}
```

- Enable `status_kind` for the `status_condition`

```
if (statuscondition->set_enabled_statuses(enabled_status_list | status_kind)
            != DDS_RETCODE_OK) {
    /* ... check for cause of failure */
}
```

- Disable `status_kind` for the `status_condition`

```
if (statuscondition->set_enabled_statuses(enabled_status_list & ~status_kind)
            != DDS_RETCODE_OK) {
    /* ... check for cause of failure */
}
```

5.46 Waitset Use Cases

Using wait-sets and conditions.

5.46.1 Setting up a wait-set

ˆ Create a wait-set

```
DDSWaitSet* waitset = new DDSWaitSet();
```

ˆ Attach conditions

```
DDSCondition* cond1 = ...;
DDSCondition* cond2 = entity->get_statuscondition();
DDSCondition* cond3 = reader->create_readcondition(DDS_NOT_READ_SAMPLE_STATE,
                                                   DDS_ANY_VIEW_STATE,
                                                   DDS_ANY_INSTANCE_STATE);
DDSCondition* cond4 = new DDSGuardCondition();
DDSCondition* cond5 = ...;
DDS_ReturnCode_t retcode;

retcode = waitset->attach_condition(cond1);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond2);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond3);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond4);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond5);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
```

5.46.2 Waiting for condition(s) to trigger

ˆ **Set up a wait-set** (p. 179)

ˆ Wait for a condition to trigger or `timeout`, whichever occurs first

```
DDS_Duration_t timeout = { 0, 1000000 }; // 1ms
```

```
DDSConditionSeq active_conditions; // holder for active conditions

bool is_cond1_triggered = false;
bool is_cond2_triggered = false;

DDS_ReturnCode_t retcode;

retcode = waitset->wait(active_conditions, timeout);

if (retcode != DDS_RETCODE_OK) {
    if (retcode == DDS_RETCODE_TIMEOUT) {
        // Timeout
    } else {
        // Failure
} else {
    // success, check if "cond1" or "cond2" are triggered:
        for(int i = 0; i < active_conditions.length(); ++i) {

           if (active_conditions[i] == cond1) {
              printf("Cond1 was triggered!");
              is_cond1_triggered = true;
           }

           if (active_conditions[i] == cond2) {
              printf("Cond2 was triggered!");
              is_cond2_triggered = true;
           }

           if (is_cond1_triggered && is_cond2_triggered) {
              break;
           }
       }
}

if (is_cond1_triggered) {
    // ... do something because "cond1" was triggered ...
}

if (is_cond2_triggered) {
    // ... do something because "cond2" was triggered ...
}

return retcode;
```

5.46.3 Tearing down a wait-set

ˆ Delete the wait-set

```
delete waitset;
waitset = NULL;
```

5.47 Transport Use Cases

Working with pluggable transports.

5.47.1 Changing the automatically registered built-in transports

- The **DDS_TRANSPORTBUILTIN_MASK_DEFAULT** (p. 397) specifies the transport plugins that will be automatically registered with a newly created **DDSDomainParticipant** (p. 1139) by default.

- This default can be changed by changing the value of the value of **TRANSPORT_BUILTIN** (p. 396) Qos Policy on the **DDSDomainParticipant** (p. 1139)

- To change the **DDS_DomainParticipantQos::transport_builtin** (p. 590) Qos Policy:

```
DDS_DomainParticipantQos participant_qos;

factory->get_default_participant_qos(participant_qos);

participant_qos.transport_builtin.mask = DDS_TRANSPORTBUILTIN_SHMEM |
                                         DDS_TRANSPORTBUILTIN_UDPv4;
```

5.47.2 Changing the properties of the automatically registered builtin transports

The behavior of the automatically registered builtin transports can be altered by changing their properties.

- Tell the **DDSDomainParticipantFactory** (p. 1216) to create the participants disabled, as described in **Turning off auto-enable of newly created participant(s)** (p. 156)

- Get the property of the desired builtin transport plugin, say ::**UDPv4 Transport** (p. 265)

```
struct NDDS_Transport_UDPv4_Property_t property = NDDS_TRANSPORT_UDPV4_PROPERTY_DEFAULT;

if (NDDSTransportSupport::get_builtin_transport_property(
                                    participant,
                                    DDS_TRANSPORTBUILTIN_UDPv4,
                                    (struct NDDS_Transport_Property_t&)property)
       != DDS_RETCODE_OK) {
    printf("***Error: get builtin transport property\n");
}
```

- Change the property fields as desired. Note that the properties should be changed carefully, as inappropriate values may prevent communications. For example, the ::**UDPv4 Transport** (p. 265) properties can be changed to support **large messages** (assuming the underlying operating system's UDPv4 stack supports the large message size). Note: if message_size_max is increased from the default for any of the built-in transports, then the **DDS_ReceiverPoolQosPolicy::buffer_size** (p. 864) on the DomainParticipant should also be changed.

```
/* Increase the UDPv4 maximum message size to 64K (large messages). */
property.parent.message_size_max =  65535;
property.recv_socket_buffer_size =  65535;
property.send_socket_buffer_size =  65535;
```

- Set the property of the desired builtin transport plugin, say ::**UDPv4 Transport** (p. 265)

```
if (NDDSTransportSupport::set_builtin_transport_property(
                                    participant,
                                    DDS_TRANSPORTBUILTIN_UDPv4,
                                    (struct NDDS_Transport_Property_t&)property)
      != DDS_RETCODE_OK) {
    printf("***Error: set builtin transport property\n");
}
```

- **Enable the participant** (p. 176) to turn on communications with other participants in the domain using the new properties for the automatically registered builtin transport plugins.

5.47.3 Creating a transport

- A transport plugin is created using methods provided by the supplier of the transport plugin.

- For example to create an instance of the ::**UDPv4 Transport** (p. 265)

```
NDDS_Transport_Plugin* transport = NULL;

struct NDDS_Transport_UDPv4_Property_t property = NDDS_TRANSPORT_UDPV4_PROPERTY_DEFAULT;

transport = NDDS_Transport_UDPv4_new(&property);

if (transport == NULL) {
   printf("***Error: creating transport plugin\n");
}
```

5.47.4 Deleting a transport

- A transport plugin can only be deleted only after the **DDSDomainParticipant** (p. 1139) with which it is registered is deleted.

- The virtual destructor provided by the abstract transport plugin API can be used to delete a transport plugin.

```
transport->delete_cEA(transport, NULL);
```

5.47.5 Registering a transport with a participant

The basic steps for setting up transport plugins for use in an RTI Connext application are described below.

- Tell the **DDSDomainParticipantFactory** (p. 1216) to create the participants disabled, as described in **Turning off auto-enable of newly created participant(s)** (p. 156)

Optionally **Changing the automatically registered built-in transports** (p. 181)

Optionally **Changing the properties of the automatically registered builtin transports** (p. 181)

- Create a disabled **DDSDomainParticipant** (p. 1139), as described in **Setting up a participant** (p. 156)

- Decide on the **network address** (p. 128) for the transport plugin. The network address should be chosen so that the resulting fully qualified address is globally unique (across all transports used in the domain).

```
/* Decide on a network address (96 bits for UDPv4), such that the fully
   qualified unicast address for the transport's interfaces will be
   globally unique. For example, we use the network address:
            1234:1234:1234:0000
   It will be prepended to the unicast addresses of the transport plugin's
   interfaces, to give a fully qualified address that is unique in the
   domain.
*/
NDDS_Transport_Address_t network_address = {{1,2,3,4, 1,2,3,4, 1,2,3,4, 0,0,0,0}};
```

- Decide on the **aliases** (p. 125) for the transport plugin. An alias can refer to one or more transport plugins. The transport class name (see **Builtin Transport Class Names** (p. 394)) are automatically appended to the user-provided aliases. Alias names are useful in creating logical groupings of transports, e.g. all the transports that are configured to support large messages may be given the alias "large_message".

```
/* Decide aliases, i.e. the names by which this transport plugin will be known */
const char* ALIASES[] = {
    "my",
    "large_message",
};
const DDS_Long ALIASES_LENGTH = sizeof(ALIASES)/sizeof(const char*);

/* Initialize the aliases StringSeq */
DDS_StringSeq aliases;
if (!aliases.from_array(ALIASES, ALIASES_LENGTH)) {
    printf("***Error: creating initializing aliases\n");
}
```

ˆ Register the transport plugin with the **DDSDomainParticipant** (p. 1139). Note that a transport plugin should **NOT be registered with more than one DomainParticipant**. It is the responsibility of the application programmer to ensure that this requirement is not violated.

```
NDDS_Transport_Handle_t handle = NDDS_TRANSPORT_HANDLE_NIL;

handle = NDDSTransportSupport::register_transport(
                        participant,          /* Disabled Domain Participant */
                        transport,            /* Transport plugin */
                        aliases,              /* Transport aliases */
                        network_address);     /* Transport network address */

if (NDDS_Transport_Handle_is_nil(&handle)) {
   printf("***Error: registering transport\n");
}
```

Optionally Adding receive routes for a transport (p. 184)

Optionally Adding send routes for a transport (p. 185)

ˆ **Enable the participant** (p. 176) to turn on communications with other participants in the domain, using the newly registered transport plugins, and automatically registered builtin transport plugins (if any).

5.47.6 Adding receive routes for a transport

ˆ Receive routes can be added to restrict address ranges on which incoming messages can be received. Any number of receive routes can be added, but these must be done before the participant is enabled.

ˆ To restrict the address range from which incoming messages can be received by the transport plugin:

```
/* Restrict to receiving messages only on interfaces
```

```
            1234:1234:1234:10.10.*.*
*/
NDDS_Transport_Address_t subnet = {{1,2,3,4, 1,2,3,4, 1,2,3,4, 10,10,0,0}};

if (NDDSTransportSupport::add_receive_route(handle, subnet, 112)
                            != DDS_RETCODE_OK) {
   printf("***Error: adding receive route\n");
}
```

5.47.7 Adding send routes for a transport

- Send routes can be added to restrict the address ranges to which outgoing messages can be sent by the transport plugin. Any number of send routes can be added, but these must be done before the participant is enabled.

- To restrict address ranges to which outgoing messages can be sent by the transport plugin:

```
/* Restrict to sending messages only to addresses (subnets)
              1234:1234:1234:10.10.30.*
*/
NDDS_Transport_Address_t subnet = {{1,2,3,4, 1,2,3,4, 1,2,3,4, 10,10,30,0}};

if (NDDSTransportSupport::add_send_route(handle, subnet, 120)
                            != DDS_RETCODE_OK) {
   printf("***Error: adding send route\n");
}
```

5.48 Filter Use Cases

Working with data filters.

5.48.1 Introduction

RTI Connext supports filtering data either during the exchange from **DDSDataWriter** (p. 1113) to **DDSDataReader** (p. 1087), or after the data has been stored at the **DDSDataReader** (p. 1087).

Filtering during the exchange process is performed by a **DDSContentFilteredTopic** (p. 1081), which is created by the **DDSDataReader** (p. 1087) as a way of specifying a subset of the data samples that it wishes to receive.

Filtering samples that have already been received by the **DDSDataReader** (p. 1087) is performed by creating a **DDSQueryCondition** (p. 1372), which can then used to check for matching samples, be alerted when matching samples arrive, or retrieve matching samples through use of the **FooDataReader::read_w_condition** (p. 1454) or **FooDataReader::take_w_condition** (p. 1456) functions. (Conditions may also be used with the APIs **FooDataReader::read_next_instance_w_condition** (p. 1468) and **FooDataReader::take_next_instance_w_condition** (p. 1470).)

Filtering may be performed on any topic, either keyed or un-keyed, except the **Built-in Topics** (p. 42). Filtering may be perfomed on any field, subset of fields, or combination of fields, subject only to the limitations of the filter syntax, and some restrictions against filtering some *sparse value types* of the **Dynamic Data** (p. 77) API.

RTI Connext contains built in support for filtering using SQL syntax, described in the **Queries and Filters Syntax** (p. 208) module.

5.48.1.1 Overview of ContentFilteredTopic

Each **DDSContentFilteredTopic** (p. 1081) is created based on an existing **DDSTopic** (p. 1419). The **DDSTopic** (p. 1419) specifies the **field_names** and **field_types** of the data contained within the topic. The **DDSContentFilteredTopic** (p. 1081), by means of its `filter_expression` and `expression_parameters`, futher specifies the *values* of the data which the **DDSDataReader** (p. 1087) wishes to receive.

Custom filters may also be constructed and utilized as described in the **Creating Custom Content Filters** (p. 191) module.

Once the **DDSContentFilteredTopic** (p. 1081) has been created, a **DDSDataReader** (p. 1087) can be created using the filtered topic. The filter's characteristics are exchanged between the **DDSDataReader** (p. 1087) and any

matching **DDSDataWriter** (p. 1113) during the discovery processs.

If the **DDSDataWriter** (p. 1113) allows (by **DDS_DataWriterResourceLimitsQosPolicy::max_remote_reader_filters** (p. 562)) and the number of filtered **DDSDataReader** (p. 1087) is less than or equal to 32, and the **DDSDataReader** (p. 1087) 's **DDS_TransportMulticastQosPolicy** (p. 978) is empty, then the **DDSDataWriter** (p. 1113) will performing filtering and send to the **DDSDataReader** (p. 1087) only those samples that meet the filtering criteria.

If disallowed by the **DDSDataWriter** (p. 1113), or if more than 32 **DDSDataReader** (p. 1087) require filtering, or the **DDSDataReader** (p. 1087) has set the **DDS_TransportMulticastQosPolicy** (p. 978), then the **DDSDataWriter** (p. 1113) sends all samples to the **DDSDataReader** (p. 1087), and the **DDSDataReader** (p. 1087) discards any samples that do not meet the filtering criteria.

Although the `filter_expression` cannot be changed once the **DDSContentFilteredTopic** (p. 1081) has been created, the `expression_parameters` can be modified using **DDSContentFilteredTopic::set_expression_parameters** (p. 1084). Any changes made to the filtering criteria by means of **DDSContentFilteredTopic::set_expression_parameters** (p. 1084), will be conveyed to any connected **DDSDataWriter** (p. 1113). New samples will be subject to the modified filtering criteria, but samples that have already been accepted or rejected are unaffected. However, if the **DDSDataReader** (p. 1087) connects to a **DDSDataWriter** (p. 1113) that *re-sends* its data, the re-sent samples will be subjected to the new filtering criteria.

5.48.1.2 Overview of QueryCondition

DDSQueryCondition (p. 1372) combine aspects of the content filtering capabilities of **DDSContentFilteredTopic** (p. 1081) with state filtering capabilities of **DDSReadCondition** (p. 1374) to create a reconfigurable means of filtering or searching data in the **DDSDataReader** (p. 1087) queue.

DDSQueryCondition (p. 1372) may be created on a disabled **DDSDataReader** (p. 1087), or after the **DDSDataReader** (p. 1087) has been enabled. If the **DDSDataReader** (p. 1087) is enabled, and has already recevied and stored samples in its queue, then all data samples in the are filtered against the **DDSQueryCondition** (p. 1372) filter criteria at the time that the **DDSQueryCondition** (p. 1372) is created. (Note that an exclusive lock is held on the **DDSDataReader** (p. 1087) sample queue for the duration of the **DDSQueryCondition** (p. 1372) creation).

Once created, incoming samples are filtered against all **DDSQueryCondition** (p. 1372) filter criteria at the time of their arrival and storage into the **DDSDataReader** (p. 1087) queue.

The number of **DDSQueryCondition** (p. 1372) filters that an individual **DDSDataReader** (p. 1087) may create is set by **DDS_DataReaderResourceLimitsQosPolicy::max_query_condition_filters** (p. 532), to an upper maximum of 32.

5.48.2 Filtering with ContentFilteredTopic

- **Set up subscriber** (p. 168)
- **Set up a topic** (p. 159)
- Create a ContentFilteredTopic, of user data type `Foo` (p. 1443):

```
DDS_ContentFilteredTopic *cft = NULL;
DDS_StringSeq parameters(2);
const char* cft_param_list[] = {"1", "100"};

cft_parameters.from_array(cft_param_list, 2);
cft = participant->create_contentfilteredtopic("ContentFilteredTopic",
                                               Foo_topic,
                                               "value > %0 AND value < %1",
                                               cft_parameters);
if (cft == NULL) {
    printf("create_contentfilteredtopic error\n");
    subscriber_shutdown(participant);
    return -1;
}
```

- Create a FooReader using the ContentFilteredTopic:

```
DDSDataReader *reader = NULL;

reader = subscriber->create_datareader(cft,
                                       datareader_qos, // or DDS_DATAREADER_QOS_DEFAULT
                                       reader_listener, // or NULL
                                       DDS_STATUS_MASK_ALL);

if (reader == NULL) {
    printf("create_datareader error\n");
    subscriber_shutdown(participant);
    return -1;
}

FooDataReader *Foo_reader = FooDataReader::narrow(reader);
if (Foo_reader == NULL) {
    printf("DataReader narrow error\n");
    subscriber_shutdown(participant);
    return -1;
}
```

Once setup, reading samples with a **DDSContentFilteredTopic** (p. 1081) is exactly the same as normal reads or takes, as decribed in **DataReader Use Cases** (p. 172).

- Changing filter crieria using set_expression_parameters:

```
cft->get_expression_parameters(cft_parameters);
DDS_String_free(cft_parameters[0]);
DDS_String_free(cft_parameters[1]);
cft_parameters[0] = DDS_String_dup("5");
cft_arameters[1] = DDS_String_dup("9");
retcode = cft->set_expression_parameters(cft_arameters);
if (retcode != DDS_RETCODE_OK) {
    printf("set_expression_parameters error\n");
    subscriber_shutdown(participant);
    return -1;
}
```

5.48.3 Filtering with Query Conditions

- Given a data reader of type **Foo** (p. 1443)

```
DDSDataReader *reader = ...;
FooDataReader *Foo_reader = FooDataReader::narrow(reader);
```

- Creating a QueryCondition

```
DDSQueryCondition *queryCondition = NULL;
DDS_StringSeq qc_parameters(2);
const char *qc_param_list[] = {"0","100"};

qc_parameters.from_array(qc_param_list,2);
queryCondition = reader->create_querycondition(DDS_NOT_READ_SAMPLE_STATE,
                                               DDS_ANY_VIEW_STATE,
                                               DDS_ALIVE_INSTANCE_STATE,
                                               "value > %0 AND value < %1",
                                               qc_parameters);

if (queryCondition == NULL) {
    printf("create_query_condition error\n");
    goto error_exit;
}
```

- Reading matching samples with a **DDSQueryCondition** (p. 1372)

```
FooSeq data_seq;
DDS_SampleInfoSeq info_seq;
retcode = Foo_reader->read_w_condition(data_seq, info_seq,
                                       DDS_LENGTH_UNLIMITED,
                                       queryCondition);

if (retcode == DDS_RETCODE_NO_DATA) {
    printf("no matching data\n");
} else if (retcode != DDS_RETCODE_OK) {
    printf("read_w_condition error %d\n", retcode);
    subscriber_shutdown(participant);
```

```
    return -1;
} else {
    for (i = 0; i < data_seq.length(); ++i) {
        if (info_seq[i].valid_data) {
            /* process your data here */
        }
    retcode = Foo_reader->return_loan(data_seq, info_seq);
    if (retcode != DDS_RETCODE_OK) {
        printf("return loan error %d\n", retcode);
        subscriber_shutdown(participant);
        return -1;
    }
}
```

- **DDSQueryCondition::set_query_parameters** (p. 1373) is used similarly to **DDSContentFilteredTopic::set_expression_parameters** (p. 1084), and the same coding techniques can be used.

- Any **DDSQueryCondition** (p. 1372) that have been created must be deleted before the **DDSDataReader** (p. 1087) can be deleted. This can be done using **DDSDataReader::delete_contained_entities** (p. 1093) or manually as in:

```
retcode = reader->delete_readcondition(queryCondition);
```

5.48.4 Filtering Performance

Although RTI Connext supports filtering on any field or combination of fields using the SQL syntax of the built-in filter, filters for keyed topics that filter solely on the contents of key fields have the potential for much higher performance. This is because for key field only filters, the **DDSDataReader** (p. 1087) caches the results of the filter (pass or not pass) for each instance. When another sample of the same instance is seen at the **DDSDataReader** (p. 1087), the filter results are retrieved from cache, dispensing with the need to call the filter function.

This optimization applies to all filtering using the built-in SQL filter, performed by the **DDSDataReader** (p. 1087), for either **DDSContentFilteredTopic** (p. 1081) or **DDSQueryCondition** (p. 1372). This does *not* apply to filtering perfomed for **DDSContentFilteredTopic** (p. 1081) by the **DDSDataWriter** (p. 1113).

5.49 Creating Custom Content Filters

Working with custom content filters.

5.49.1 Introduction

By default, RTI Connext creates content filters with the DDS_SQL_FILTER, which implements a superset of the DDS-specified SQL WHERE clause. However, in many cases this filter may not be what you want. Some examples are:

- The default filter can only filter based on the content of a sample, not on a computation on the content of a sample. You can use a custom filter that is customized for a specific type and can filter based on a computation of the type members.
- You want to use a different filter language then SQL

This HOWTO explains how to write your own custom filter and is divided into the following sections:

- **The Custom Content Filter API** (p. 191)
- **Example Using C format strings** (p. 192)

5.49.2 The Custom Content Filter API

A custom content filter is created by calling the **DDSDomainParticipant::register_contentfilter** (p. 1156) function with a **DDSContentFilter** (p. 1077) that contains a **compile**, an **evaluate** function and a **finalize** function. **DDSContentFilteredTopic** (p. 1081) can be created with **DDSDomainParticipant::create_contentfilteredtopic_with_filter** (p. 1180) to use this filter.

A custom content filter is used by RTI Connext at the following times during the life-time of a **DDSContentFilteredTopic** (p. 1081) (the function called is shown in parenthesis).

- When a **DDSContentFilteredTopic** (p. 1081) is created (**compile** (p. 192))
- When the filter parameters are changed on the **DDSContentFilteredTopic** (p. 1081) (**compile** (p. 192)) with **DDSContentFilteredTopic::set_expression_parameters** (p. 1084)

- When a sample is filtered (**evaluate** (p. 192)). This function is called by the RTI Connext core with a de-serialized sample
- When a **DDSContentFilteredTopic** (p. 1081) is deleted (**finalize** (p. 192))

5.49.2.1 The compile function

The **compile** (p. 192) function is used to **compile** a filter expression and expression parameters. Please note that the term **compile** is intentionally loosely defined. It is up to the user to decide what this function should do and return.

See **DDSContentFilter::compile** (p. 1078) for details.

5.49.2.2 The evaluate function

The **evaluate** (p. 193) function is called each time a sample is received to determine if a sample should be filtered out and discarded.

See **DDSContentFilter::evaluate** (p. 1079) for details.

5.49.2.3 The finalize function

The **finalize** (p. 193) function is called when an instance of the custom content filter is no longer needed. When this function is called, it is safe to free all resources used by this particular instance of the custom content filter.

See **DDSContentFilter::finalize** (p. 1080) for details.

5.49.3 Example Using C format strings

Assume that you have a type **Foo** (p. 1443).

You want to write a custom filter function that will drop all samples where the value of Foo.x > x and x is a value determined by an expression parameter. The filter will **only** be used to filter samples of type **Foo** (p. 1443).

5.49.3.1 Writing the Compile Function

The first thing to note is that we can ignore the filter expression, since we already know what the expression is. The second is that x is a parameter that can be changed. By using this information, the compile function is very easy to implement. Simply return the parameter string. This string will then be passed to the evaluate function every time a sample of this type is filtered.

Below is the entire **compile** (p. 192) function.

```
DDS_ReturnCode_t MyContentFilter::compile(
    void** new_compile_data, const char * /* expression */,
    const DDS_StringSeq& parameters,const DDS_TypeCode* /* type_code */,
    const char * /* type_class_name */,
    void * /* old_compile_data */) {

    *new_compile_data = (void*)DDS_String_dup(parameters[0]);

    return DDS_RETCODE_OK;
}
```

5.49.3.2 Writing the Evaluate Function

The next step is to implement the **evaluate** function. The evaluate function receives the parameter string with the actual value to test against. Thus the evaluate function must read the actual value from the parameter string before evaluating the expression. Below is the entire **evaluate** (p. 192) function.

```
DDS_Boolean MyContentFilter::evaluate(
    void* compile_data,const void* sample,const struct DDS_FilterSampleInfo * meta_data) {

    char *parameter = (char*)compile_data;
    DDS_Long x;

    Foo *foo_sample = (Foo*)sample;

    sscanf(parameter,"%d",&x);

    return (foo_sample->x > x ? DDS_BOOLEAN_FALSE : DDS_BOOLEAN_TRUE);
}
```

5.49.3.3 Writing the Finalize Function

The last function to write is the finalize function. It is safe to free all resources used by this particular instance of the custom content filter that is allocated in **compile**. Below is the entire **finalize** (p. 192) function.

```
void MyContentFilter::finalize(
    void* compile_data) {
    /* free parameter string from compile function */
    DDS_String_free((char *)compile_data);
}
```

5.49.3.4 Registering the Filter

Before the custom filter can be used, it must be registered with RTI Connext:

```
DDSDomainParticipant *myParticipant=NULL;

/* myParticipant = .... */

DDSContentFilter *myCustomFilter = new MyContentFilter();

if (myParticipant->register_contentfilter(
    (char*)"MyCustomFilter",
    myCustomFilter) != DDS_RETCODE_OK) {
    printf("Failed to register custom filter\n");
}
```

5.49.3.5 Unregistering the Filter

When the filter is no longer needed, it can be unregistered from RTI Connext:

```
DDSDomainParticipant *myParticipant = NULL;

/* myParticipant = .... */

DDSContentFilter *myCustomFilter =
    myParticipant->lookup_contentfilter((char*)"MyCustomFilter");

if (myCustomFilter != NULL) {
    if (myParticipant->unregister_contentfilter(
        (char*)"MyCustomFilter") != DDS_RETCODE_OK) {
        printf("Failed to unregister custom filter\n");
    }

    delete myCustomFilter;
}
```

5.50 Large Data Use Cases

Working with large data types.

5.50.1 Introduction

RTI Connext supports data types whose size exceeds the maximum message size of the underlying transports. A **DDSDataWriter** (p. 1113) will fragment data samples when required. Fragments are automatically reassembled at the receiving end.

Once all fragments of a sample have been received, the new sample is passed to the **DDSDataReader** (p. 1087) which can then make it available to the user. Note that the new sample is treated as a regular sample at that point and its availability depends on standard QoS settings such as **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) and **DDS_KEEP_LAST_HISTORY_QOS** (p. 368).

The large data feature is fully supported by all DDS API's, so its use is mostly transparent. Some additional considerations apply as explained below.

5.50.2 Writing Large Data

In order to use the large data feature with the **DDS_RELIABLE_RELIABILITY_QOS** (p. 363) setting, the **DDSDataWriter** (p. 1113) must be configured as an asynchronous writer (**DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422)) with associated **DDSFlowController** (p. 1259).

While the use of an asynchronous writer and flow controller is optional when using the **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363) setting, most large data use cases will benefit from the use of a flow controller to prevent flooding the network when fragments are being sent.

- **Set up writer** (p. 166)
- **Add flow control** (p. 161)

5.50.3 Receiving Large Data

Large data is supported by default and in most cases, no further changes are required.

The **DDS_DataReaderResourceLimitsQosPolicy** (p. 521) allows tuning

the resources available to the **DDSDataReader** (p. 1087) for reassembling fragmented large data.

- **Set up reader** (p. 172)

5.51 Documentation Roadmap

This section contains a roadmap for the new user with pointers on what to read first.

If you are new to RTI Connext, we recommend starting in the following order:

- See the `Getting Started Guide`. This document provides download and installation instructions. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application.

- The `User's Manual` describes the features of the product and how to use them. It is organized around the structure of the DDS APIs and certain common high-level tasks.

- The documentation in the **DDS API Reference** (p. 203) provides an overview of API classes and modules for the DDS data-centric publish-subscribe (DCPS) package from a programmer's perspective. Start by reading the documentation on the main page.

- After reading the high level module documentation, look at the **Publication Example** (p. 154) and **Subscription Example** (p. 155) for step-by-step examples of creating a publication and subscription. These are hyperlinked code snippets to the full API documentation, and provide a good place to begin learning the APIs.

- Next, work through your own application using the example code files generated by **rtiddsgen** (p. 220).

- To integrate similar code into your own application and build system, you will likely need to refer to the `Platform Notes`.

5.52 Conventions

This section describes the conventions used in the API documentation.

5.52.1 Unsupported Features

[**Not supported (optional)**] This note means that the optional feature from the DDS specification is not supported in the current release.

5.52.2 API Naming Conventions

5.52.2.1 Structure & Class Names

RTI Data Distribution Service 4 makes a distinction between *value types* and *interface types*. Value types are types such as primitives, enumerations, strings, and structures whose identity and equality are determined solely by explicit state. Interface types are those abstract opaque data types that conceptually have an identity apart from their explicit state. Examples include all of the **DDSEntity** (p. 1253) subtypes, the **DDSCondition** (p. 1075) subtypes, and **DDSWaitSet** (p. 1433). Instances of value types are frequently transitory and are declared on the stack. Instances of interface types typically have longer lifecycles, are accessible by pointer only, and may be managed by a factory object.

Value and interface types are distinguished by their names: value types have names beginning with "DDS_" (i.e. with an underscore); interface types have names beginning with "DDS" (i.e. with no underscore). Another way to think of it: C-style types – structures, enumerations, etc. – have names beginning with "DDS_"; C++ classes have names beginning with "DDS."

5.52.3 API Documentation Terms

In the API documentation, the term module refers to a logical grouping of documentation and elements in the API.

At this time, typedefs that occur in the API, such as **DDS_ReturnCode_t** (p. 315) do not show up in the compound list or indices. This is a known limitation in the generated HTML.

5.52.4 Stereotypes

Commonly used stereotypes in the API documentation include the following.

5.52.4.1 Extensions

ˆ *<<**eXtension**>> (p. 199)*

- An RTI Connext product extension to the DDS standard specification.
- The extension APIs complement the standard APIs specified by the OMG DDS specification. They are provided to improve product usability and enable access to product-specific features such as pluggable transports.

5.52.4.2 Experimental

ˆ *<<**experimental**>> (p. 199)*

- RTI Connext experimental features are used to evaluate new features and get user feedback.
- These features are not guaranteed to be fully supported and might be implemented only of some of the programming languages supported bt RTI Connext
- The functional APIs corresponding to experimental features can be distinguished from other APIs by the suffix '_exp'.
- Experimental features may or may not appear in future product releases.
- The name of the experimental features APIs will change if they become officially supported. At the very least the suffix '_exp' will be removed.
- Experimental features should not be used in production.

5.52.4.3 Types

ˆ *<<**interface**>> (p. 199)*

- Pure interface type with *no state*.
- Languages such as Java natively support the concept of an *interface* type, which is a collection of method signatures devoid of any dynamic state.
- In C++, this is achieved via a class with all *pure virtual* methods and devoid of any instance variables (ie no dynamic state).
- Interfaces are generally organized into a type hierarchy. Static typecasting along the interface type hierarchy is "safe" for valid objects.

ˆ *<<**generic**>> (p. 199)*

- A *generic* type is a *skeleton* class written in terms of generic parameters. Type-specific instantiations of such types are conventionally referred to in this documentation in terms of the hypothetical type "Foo"; for example: **FooSeq** (p. 1494), FooDataType, **FooDataWriter** (p. 1475), and **FooDataReader** (p. 1444).
- For portability and efficiency, we implement generics using C preprocessor macros, rather than using C++ templates.
- A *generic* type interface is declared via a #define macro.
- Concrete types are generated from the generic type statically at compile time. The implementation of the concrete types is provided via the generic macros which can then be compiled as normal C or C++ code.

ˆ <<***singleton***>> *(*p. *200)*

- Singleton class. There is a single instance of the class.
- Generally acccessed via a `get_instance()` static method.

5.52.4.4 Method Parameters

ˆ <<***in***>> *(*p. *200)*

- An *input* parameter.

ˆ <<***out***>> *(*p. *200)*

- An *output* parameter.

ˆ <<***inout***>> *(*p. *200)*

- An *input* and *output* parameter.

5.53 Using DDS:: Namespace

This section describes the C++ namespace support in the DDS API.

5.53.1 DDS Namespace Support

In this documentation, all C++ classes, value types, interface types and constants have names beginning with either **"DDS_"** or **"DDS"**. Alternatively, **DDS** namespace can also be used to refer to all these classes, types or constant.

All the C++ API that begins with either **"DDS_"** or **"DDS"** can be replaced with its namespace equivalent. For example, **DDSDomainParticipant** (p. 1139) has a namespace equivalent of **DDS::DomainParticipant**, and **DDS_DomainParticipantQos** (p. 588) has a namespace equivalent of **DDS::DomainParticipantQos**.

In order to use the **DDS** namespace, an additional header file, **ndds_namespace_cpp.h** , will need to be included in your source file:

```
#include "ndds/ndds_cpp.h"
#include "ndds/ndds_namespace_cpp.h"

DDS::DomainParticipant *participant = NULL;
DDS::DomainParticipantQos participant_qos;
DDS::DomainParticipantListener *listener = NULL;
DDS::StatusKind status_kind = DDS::INCONSISTENT_TOPIC_STATUS;
DDS::Long counter = 0L;
```

If the namespace header file is not included in the source file, DDS namespace cannot be used in the RTI Connext API.

5.53.2 DDS Namespace and Primitive Types

By default, **DDS** namespace support for primitive types are included. With **DDS** namespace support, the difference between DDS types and native types can just be the capitalization in some cases:

```
#include "ndds/ndds_cpp.h"
#include "ndds/ndds_namespace_cpp.h"

using namespace DDS;

Long ddsCounter = 0L;
long nativeCounter = 0L;
```

If you want to exclude the DDS namespace support for primitive types, you can define **NDDS_EXCLUDE_PRIMITIVE_TYPES_FROM_-**

NAMESPACE in you application before including the namespace header file. **DDS** namespace support for primitive types will then be excluded:

```
#include "ndds/ndds_cpp.h"
#define NDDS_EXCLUDE_PRIMITIVE_TYPES_FROM_NAMESPACE
#include "ndds/ndds_namespace_cpp.h"

using namespace DDS;

DDS_Long ddsCounter = 0;
long nativeCounter = 0;
```

For the rest of the documentation, the **DDS** prefix is used for all class /types/constants names. However, all the API with the **DDS** prefix can be replaced with **DDS** namespace instead if the namespace header file is included.

5.54 DDS API Reference

RTI Connext modules following the DDS module definitions.

Modules

- **Domain Module**

  *Contains the **DDSDomainParticipant** (p. 1139) class that acts as an entrypoint of RTI Connext and acts as a factory for many of the classes. The **DDSDomainParticipant** (p. 1139) also acts as a container for the other objects that make up RTI Connext.*

- **Topic Module**

  *Contains the **DDSTopic** (p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322) classes, the **DDSTopicListener** (p. 1430) interface, and more generally, all that is needed by an application to define **DDSTopic** (p. 1419) objects and attach QoS policies to them.*

- **Publication Module**

  *Contains the **DDSFlowController** (p. 1259), **DDSPublisher** (p. 1346), and **DDSDataWriter** (p. 1113) classes as well as the **DDSPublisherListener** (p. 1370) and **DDSDataWriterListener** (p. 1133) interfaces, and more generally, all that is needed on the publication side.*

- **Subscription Module**

  *Contains the **DDSSubscriber** (p. 1390), **DDSDataReader** (p. 1087), **DDSReadCondition** (p. 1374), and **DDSQueryCondition** (p. 1372) classes, as well as the **DDSSubscriberListener** (p. 1414) and **DDSDataReaderListener** (p. 1108) interfaces, and more generally, all that is needed on the subscription side.*

- **Infrastructure Module**

  Defines the abstract classes and the interfaces that are refined by the other modules. Contains common definitions such as return codes, status values, and QoS policies.

- **Queries and Filters Syntax**

5.54.1 Detailed Description

RTI Connext modules following the DDS module definitions.

5.54.2 Overview

Information flows with the aid of the following constructs: **DDSPublisher** (p. 1346) and **DDSDataWriter** (p. 1113) on the sending side, **DDSSubscriber** (p. 1390) and **DDSDataReader** (p. 1087) on the receiving side.

- A **DDSPublisher** (p. 1346) is an object responsible for data distribution. It may publish data of different data types. A TDataWriter acts as a *typed* (i.e. each **DDSDataWriter** (p. 1113) object is dedicated to one application data type) accessor to a publisher. A **DDSDataWriter** (p. 1113) is the object the application must use to communicate to a publisher the existence and value of data objects of a given type. When data object values have been communicated to the publisher through the appropriate data-writer, it is the publisher's responsibility to perform the distribution (the publisher will do this according to its own QoS, or the QoS attached to the corresponding data-writer). A *publication* is defined by the association of a data-writer to a publisher. This association expresses the intent of the application to publish the data described by the data-writer in the context provided by the publisher.

- A **DDSSubscriber** (p. 1390) is an object responsible for receiving published data and making it available (according to the Subscriber's QoS) to the receiving application. It may receive and dispatch data of different specified types. To access the received data, the application must use a *typed* TDataReader attached to the subscriber. Thus, a *subscription* is defined by the association of a data-reader with a subscriber. This association expresses the intent of the application to subscribe to the data described by the data-reader in the context provided by the subscriber.

DDSTopic (p. 1419) objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A **DDSTopic** (p. 1419) is meant to fulfill that purpose: it associates a name (unique in the domain i.e. the set of applications that are communicating with each other), a data type, and QoS related to the data itself. In addition to the topic QoS, the QoS of the **DDSDataWriter** (p. 1113) associated with that Topic and the QoS of the **DDSPublisher** (p. 1346) associated to the **DDSDataWriter** (p. 1113) control the behavior on the publisher's side, while the corresponding **DDSTopic** (p. 1419), **DDSDataReader** (p. 1087) and **DDSSubscriber** (p. 1390) QoS control the behavior on the subscriber's side.

When an application wishes to publish data of a given type, it must create a **DDSPublisher** (p. 1346) (or reuse an already created one) and a **DDSDataWriter** (p. 1113) with all the characteristics of the desired publication. Similarly, when an application wishes to receive data, it must create a **DDSSub-**

scriber (p. 1390) (or reuse an already created one) and a **DDSDataReader** (p. 1087) to define the subscription.

5.54.3 Conceptual Model

The overall conceptual model is shown below.

Notice that all the main communication objects (the specializations of Entity) follow unified patterns of:

- Supporting QoS (made up of several QosPolicy); QoS provides a generic mechanism for the application to control the behavior of the Service and tailor it to its needs. Each **DDSEntity** (p. 1253) supports its own specialized kind of QoS policies (see **QoS Policies** (p. 331)).

- Accepting a **DDSListener** (p. 1318); listeners provide a generic mechanism for the middleware to notify the application of relevant asynchronous events, such as arrival of data corresponding to a subscription, violation of a QoS setting, etc. Each **DDSEntity** (p. 1253) supports its own specialized kind of listener. Listeners are related to changes in status conditions (see **Status Kinds** (p. 317)).

  Note that only one Listener per entity is allowed (instead of a list of them). The reason for that choice is that this allows a much simpler (and, thus, more efficient) implementation as far as the middleware is concerned. Moreover, if it were required, the application could easily implement a listener that, when triggered, triggers in return attached 'sub-listeners'.

- Accepting a **DDSStatusCondition** (p. 1376) (and a set of **DDSReadCondition** (p. 1374) objects for the **DDSDataReader** (p. 1087)); conditions (in conjunction with **DDSWaitSet** (p. 1433) objects) provide support for an alternate communication style between the middleware and the application (i.e., wait-based rather than notification-based).

All DCPS entities are attached to a **DDSDomainParticipant** (p. 1139). A domain participant represents the local membership of the application in a domain. A *domain* is a distributed concept that links all the applications able to communicate with each other. It represents a communication plane: only the publishers and the subscribers attached to the same domain may interact.

DDSDomainEntity (p. 1138) is an intermediate object whose only purpose is to state that a DomainParticipant cannot contain other domain participants.

At the DCPS level, data types represent information that is sent atomically. For performance reasons, only plain data structures are handled by this level.

By default, each data modification is propagated individually, independently, and uncorrelated with other modifications. However, an application may request that several modifications be sent as a whole and interpreted as such at the recipient side. This functionality is offered on a Publisher/Subscriber basis. That is, these relationships can only be specified among **DDSDataWriter** (p. 1113) objects attached to the same **DDSPublisher** (p. 1346) and retrieved among **DDSDataReader** (p. 1087) objects attached to the same **DDSSubscriber** (p. 1390).

By definition, a **DDSTopic** (p. 1419) corresponds to a single data type. However, several topics may refer to the same data type. Therefore, a **DDSTopic** (p. 1419) identifies data of a single type, ranging from one single instance to a whole collection of instances of that given type. This is shown below for the hypothetical data type `Foo` (p. 1443).

In case a set of instances is gathered under the same topic, different instances must be distinguishable. This is achieved by means of the values of some data fields that form the **key** to that data set. The *key description* (i.e., the list of data fields whose value forms the key) has to be indicated to the middleware. The rule is simple: *different data samples with the same key value represent successive values for the same instance, while different data samples with different key values represent different instances*. If no key is provided, the data set associated with the **DDSTopic** (p. 1419) is restricted to a *single instance*.

Topics need to be known by the middleware and potentially propagated. Topic objects are created using the create operations provided by **DDSDomainParticipant** (p. 1139).

The interaction style is straightforward on the publisher's side: when the application decides that it wants to make data available for publication, it calls the appropriate operation on the related **DDSDataWriter** (p. 1113) (this, in turn, will trigger its **DDSPublisher** (p. 1346)).

On the subscriber's side however, there are more choices: relevant information may arrive when the application is busy doing something else or when the application is just waiting for that information. Therefore, depending on the way the application is designed, asynchronous notifications or synchronous access may be more appropriate. Both interaction modes are allowed, a **DDSListener** (p. 1318) is used to provide a callback for synchronous access and a **DDSWaitSet** (p. 1433) associated with one or several **DDSCondition** (p. 1075) objects provides asynchronous data access.

The same synchronous and asynchronous interaction modes can also be used to access changes that affect the middleware communication status (see **Status Kinds** (p. 317)). For instance, this may occur when the middleware asynchronously detects an inconsistency. In addition, other middleware information that may be relevant to the application (such as the list of the existing topics) is made available by means of **built-in topics** (p. 42) that the application can access as plain application data, using built-in data-readers.

5.54.4 Modules

DCPS consists of five modules:

- **Infrastructure module** (p. 118) defines the abstract classes and the interfaces that are refined by the other modules. It also provides support for the two interaction styles (notification-based and wait-based) with the middleware.

- **Domain module** (p. 32) contains the **DDSDomainParticipant** (p. 1139) class that acts as an entrypoint of the Service and acts as a factory for many of the classes. The **DDSDomainParticipant** (p. 1139) also acts as a container for the other objects that make up the Service.

- **Topic module** (p. 49) contains the **DDSTopic** (p. 1419) class, the **DDSTopicListener** (p. 1430) interface, and more generally, all that is needed by the application to define **DDSTopic** (p. 1419) objects and attach QoS policies to them.

- **Publication module** (p. 82) contains the **DDSPublisher** (p. 1346) and **DDSDataWriter** (p. 1113) classes as well as the **DDSPublisherListener** (p. 1370) and **DDSDataWriterListener** (p. 1133) interfaces, and more generally, all that is needed on the publication side.

- **Subscription module** (p. 95) contains the **DDSSubscriber** (p. 1390), **DDSDataReader** (p. 1087), **DDSReadCondition** (p. 1374), and **DDSQueryCondition** (p. 1372) classes, as well as the **DDSSubscriberListener** (p. 1414) and **DDSDataReaderListener** (p. 1108) interfaces, and more generally, all that is needed on the subscription side.

5.55 Queries and Filters Syntax

5.55.1 Syntax for DDS Queries and Filters

A subset of SQL syntax is used in several parts of the specification:

- The `filter_expression` in the **DDSContentFilteredTopic** (p. 1081)
- The `query_expression` in the **DDSQueryCondition** (p. 1372)
- The `topic_expression` in the **DDSMultiTopic** (p. 1322)

Those expressions may use a subset of SQL, extended with the possibility to use program variables in the SQL expression. The allowed SQL expressions are defined with the BNF-grammar below.

The following notational conventions are made:

- *NonTerminals* are typeset in italics.
- `'Terminals'` are quoted and typeset in a fixed width font. They are written in upper case in most cases in the BNF-grammar below, but should be case insensitive.
- **TOKENS** are typeset in bold.
- The notation (*element* // ',') represents a non-empty comma-separated list of *elements*.

5.55.2 SQL grammar in BNF

```
Expression ::= FilterExpression
             | TopicExpression
             | QueryExpression
             .
FilterExpression ::= Condition
TopicExpression  ::= SelectFrom { Where } ';'
QueryExpression  ::= { Condition }{ 'ORDER BY' ( FIELD-
NAME // ',') }
                   .

SelectFrom       ::= 'SELECT' Aggregation 'FROM' Selection
                   .
Aggregation      ::= '*'
                   | ( SubjectFieldSpec // ',')
                   .
SubjectFieldSpec ::= FIELDNAME
```

```
              |    FIELDNAME 'AS' IDENTIFIER
              |    FIELDNAME IDENTIFIER
              .
Selection   ::= TOPICNAME
          |   TOPICNAME NaturalJoin JoinItem
          .
JoinItem    ::= TOPICNAME
          |   TOPICNAME NaturalJoin JoinItem
          |   '(' TOPICNAME NaturalJoin JoinItem ')'
          .
NaturalJoin ::= 'INNER JOIN'
          |   'INNER NATURAL JOIN'
          |   'NATURAL JOIN'
          |   'NATURAL INNER JOIN'
          .
Where       ::= 'WHERE' Condition
          .
Condition   ::= Predicate
          |   Condition 'AND' Condition
          |   Condition 'OR' Condition
          |   'NOT' Condition
          |   '(' Condition ')'
          .
Predicate   ::= ComparisonPredicate
          |   BetweenPredicate
          .
ComparisonPredicate ::= ComparisonTerm RelOp ComparisonTerm
                  .
ComparisonTerm      ::= FieldIdentifier
                  | Parameter
                  .
BetweenPredicate    ::= FieldIdentifier 'BETWEEN' Range
                  |   FieldIdentifier 'NOT BETWEEN' Range
                  .
FieldIdentifier       ::= FIELDNAME
                  | IDENTIFIER
                  .
RelOp     ::= '=' | '>' | '>=' | '<' | '<=' | '<>' | 'LIKE' | 'MATCH'
        .
Range     ::= Parameter 'AND' Parameter
        .
Parameter ::= INTEGERVALUE
        |   CHARVALUE
        |   FLOATVALUE
        |   STRING
        |   ENUMERATEDVALUE
        |   BOOLEANVALUE
        |   PARAMETER
        .
```

Note – INNER JOIN, INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. They are all supported because they all are part of the SQL standard.

5.55.3 Token expression

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

- **IDENTIFIER** - An identifier for a FIELDNAME, and is defined as any series of characters 'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '_' but may not start with a digit.

Formal notation:

```
IDENTIFIER:  LETTER ( PART_LETTER)*
where  LETTER: [ "A"-"Z","_","a"-"z" ]
       PART_LETTER: [ "A"-"Z","_","a"-"z","0"-"9" ]
```

- **FIELDNAME** - A fieldname is a reference to a field in the data structure. The dot '.' is used to navigate through nested structures. The number of dots that may be used in a FIELDNAME is unlimited. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure, which may or may not match the fieldnames that appear on the language-specific (e.g., C/C++, Java) mapping of the structure. To reference to the n+1 element in an array or sequence, use the notation '[n]', where n is a natural number (zero included). FIELDNAME must resolve to a primitive IDL type; that is either boolean, octet, (unsigned) short, (unsigned) long, (unsigned) long long, float double, char, wchar, string, wstring, or enum.

Formal notation:

```
FIELDNAME:  FieldNamePart ( "."  FieldNamePart )*
where  FieldNamePart : IDENTIFIER ( "["  Index "]" )*
       Index> : (["0"-"9"])+
              | ["0x","0X"](["0"-"9", "A"-"F", "a"-"f"])+
```

Primitive IDL types referenced by FIELDNAME are treated as different types in *Predicate* according to the following table:

Predicate Data Type	IDL Type
BOOLEANVALUE	boolean
INTEGERVALUE	octet, (unsigned) short, (unsigned) long, (unsigned) long long
FLOATVALUE	float, double
CHARVALUE	char, wchar
STRING	string, wstring
ENUMERATEDVALUE	enum

- **TOPICNAME** - A topic name is an identifier for a topic, and is defined as any series of characters 'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '_' but may not start with a digit.

Formal notation:

```
TOPICNAME : IDENTIFIER
```

- **INTEGERVALUE** - Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. A hexadecimal number is preceded by 0x and must be a valid hexadecimal expression.

Formal notation:

```
INTEGERVALUE : (["+","-"])? (["0"-"9"])+ [("L","l")]?
             | (["+","-"])? ["0x","0X"](["0"-"9", "A"-"F", "a"-"f"])+ [("L","l")]?
```

- **CHARVALUE** - A single character enclosed between single quotes.

Formal notation:

```
CHARVALUE : "'" (~["'"])? "'"
```

- **FLOATVALUE** - Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point ('.'). A power-of-ten expression may be postfixed, which has the syntax en or En, where n is a number, optionally preceded by a plus or minus sign.

Formal notation:

```
FLOATVALUE : (["+","-"])? (["0"-"9"])* (".")? (["0"-"9"])+ ( EXPONENT)?
where  EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+
```

- **STRING** - Any series of characters encapsulated in single quotes, except the single quote itself.

Formal notation:

```
STRING : "'" (~["'"])* "'"
```

- **ENUMERATEDVALUE** - An enumerated value is a reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.

Formal notation:

```
ENUMERATEDVALUE : "'" ["A" - "Z", "a" - "z"] ["A" - "Z", "a" - "z", "_", "0" - "9"]* "'"
```

- **BOOLEANVALUE** - Can either be 'TRUE' or 'FALSE', case insensitive.

Formal notation (case insensitive):

```
BOOLEANVALUE : ["TRUE","FALSE"]
```

- **PARAMETER** - A parameter is of the form $\%n$, where n represents a natural number (zero included) smaller than 100. It refers to the n + 1 $^{\text{th}}$ argument in the given context. Argument can only in primitive type value format. It cannot be a FIELDNAME.

Formal notation:

```
PARAMETER : "%" (["0"-"9"])+
```

5.55.4 String Parameters

Strings used as parameter values must contain the enclosing quotation marks (') within the parameter value, and not place the quotation marks within the expression statement. For example, the following expression is legal:

```
" symbol MATCH %0 " with parameter 0 = " 'IBM' "
```

whereas the following expression will not compile:

```
" symbol MATCH '%0' " with parameter 0 = " IBM "
```

5.55.5 Type compatability in Predicate

Only certain combination of type comparisons are valid in *Predicate*. The following table marked all the compatible pairs with 'YES':

	BOOLEAN-VALUE	**INTE-GERVALUE**	**FLOAT-VALUE**	**CHAR-VALUE**	**STRING**	**ENU-MER-ATED-VALUE**
BOOLEAN	YES					
INTE-GERVALUE		YES	YES			
FLOAT-VALUE		YES	YES			
CHAR-VALUE				YES	YES	YES
STRING				YES	YES(∗1)	YES
ENU-MER-ATED-VALUE		YES		YES(∗2)	YES(∗2)	YES(∗3)

- (∗1) See **SQL Extension: Regular Expression Matching** (p. 213)
- (∗2) Because the formal notation of the Enumeration values, they are compatible with string and char literals, but they are not compatible with string or char variables, i.e., "MyEnum='EnumValue'" would be correct, but "MyEnum=MyString" is not allowed.
- (∗3) Only for same type Enums.

5.55.6 SQL Extension: Regular Expression Matching

The relational operator MATCH may only be used with string fields. The right-hand operator is a string *pattern*. A string pattern specifies a template that the left-hand field value must match. The characters ,/?∗[]-^!% have special meanings.

MATCH is case-sensitive.

The pattern allows limited "wild card" matching under the following rules:

Character	Meaning
,	"," separates a list of alternate patterns. The field string is matched if it matches one or more of the patterns.
/	"/" in the pattern string matches a / in the field string. This character is used to separate a sequence of mandatory substrings.
?	"?" in the pattern string matches any single *non-special* characters in the field string.
*	"*" in the pattern string matches 0 or more *non-special* characters in field string.
[***charlist***]	Matches any one of the characters from the list of characters in *charlist*.
[***s-e***]	Matches any character any character from *s* to *e*, inclusive.
%	"%" is used to designate filter expressions parameters.
[**!*charlist***] **or** [^***charlist***]	Matches any characters not in *charlist* (not supported).
[**!*s-e***] **or** [^***s-e***]	Matches any characters not in the interval [s-e] (not supported).
\	Escape character for special characters (not supported)

The syntax is similar to the POSIX fnmatch syntax (1003.2-1992 section B.6). The MATCH syntax is also similar to the 'subject' strings of TIBCO Rendezvous.

5.55.7 Examples

Assuming Topic "Location" has as an associated type a structure with fields "flight_id, x, y, z", and Topic "FlightPlan" has as fields "flight_id, source, destination". The following are examples of using these expressions.

Example of a **filter_expression** (for **DDSContentFilteredTopic** (p. 1081)) or a **query_expression** (for **DDSQueryCondition** (p. 1372)):

```
^  "z < 1000 AND x < 23"
```

Examples of a **filter_expression** using **MATCH** (for **DDSContentFilteredTopic** (p. 1081)) operator:

- "symbol MATCH 'NASDAQ/GOOG'"
- "symbol MATCH 'NASDAQ/[A-M]*'"

Example of a **topic_expression** (for **DDSMultiTopic** (p. 1322) [**Not supported (optional)**]):

- ```
  "SELECT flight_id, x, y, z AS height FROM 'Location' NATURAL JOIN
        'FlightPlan' WHERE height < 1000 AND x <23"
  ```
  ```

5.56 RTI Connext API Reference

RTI Connext product specific API's.

Modules

- **Clock Selection**

  APIs related to clock selection.

- **Multi-channel DataWriters**

  APIs related to Multi-channel DataWriters.

- **Pluggable Transports**

  APIs related to RTI Connext pluggable transports.

- **Configuration Utilities**

  Utility API's independent of the DDS standard.

- **Unsupported Utilities**

  Unsupported APIs used by examples in the RTI Connext distribution as well as in rtiddsgen-generated examples.

- **Durability and Persistence**

  APIs related to RTI Connext Durability and Persistence.

- **System Properties**

  System Properties.

- **Configuring QoS Profiles with XML**

  APIs related to XML QoS Profiles.

5.56.1 Detailed Description

RTI Connext product specific API's.

5.57 Programming How-To's

These "How To"s illustrate how to apply RTI Connext APIs to common use cases.

Modules

- **Publication Example**
  A data publication example.
- **Subscription Example**
  A data subscription example.
- **Participant Use Cases**
  Working with domain partcipants.
- **Topic Use Cases**
  Working with topics.
- **FlowController Use Cases**
  Working with flow controllers.
- **Publisher Use Cases**
  Working with publishers.
- **DataWriter Use Cases**
  Working with data writers.
- **Subscriber Use Cases**
  Working with subscribers.
- **DataReader Use Cases**
  Working with data readers.
- **Entity Use Cases**
  Working with entities.
- **Waitset Use Cases**
  Using wait-sets and conditions.
- **Transport Use Cases**
  Working with pluggable transports.

- **Filter Use Cases**

  Working with data filters.

- **Creating Custom Content Filters**

  Working with custom content filters.

- **Large Data Use Cases**

  Working with large data types.

5.57.1 Detailed Description

These "How To"s illustrate how to apply RTI Connext APIs to common use cases.

These are a good starting point to familiarize yourself with DDS. You can use these code fragments as "templates" for writing your own code.

5.58 Programming Tools

Modules

- **rtiddsgen**

  Generates source code from data types declared in IDL, XML, XSD, or WSDL files.

- **rtiddsping**

  Sends or receives simple messages using RTI Connext.

- **rtiddsspy**

  Debugging tool which receives all RTI Connext communication.

5.59 rtiddsgen

Generates source code from data types declared in IDL, XML, XSD, or WSDL files. Generates code necessary to allocate, send, receive, and print user-defined data types.

5.59.1 Usage

```
rtiddsgen [-d <outdir>]
     [-language <C|C++|Java|C++/CLI|C#|Ada>]
             [-namespace]
             [-package <packagePrefix>]
             [-example <arch>]
             [-replace]
     [-debug]
             [-corba [client header file] [-orb <CORBA ORB>]]
             [-optimization <level of optimization>]
             [-stringSize <Unbounded strings size>]
             [-sequenceSize <Unbounded sequences size>]
             [-notypecode]
             [-ppDisable]
             [-ppPath <preprocessor executable>]
             [-ppOption <option>]
             [-D <name>[=<value>]]
             [-U <name>]
             [-I <directory>]
             [-noCopyable]
             [-use42eAlignment]
             [-enableEscapeChar]
             [-typeSequenceSuffix <Suffix>]
             [-dataReaderSuffix <Suffix>]
             [-dataWriterSuffix <Suffix>]
             [-convertToXml |
              -convertToXsd |
              -convertToWsdl |
              -convertToIdl]
             [-convertToCcl]
             [-convertToCcs]
             [-expandOctetSeq]
             [-expandCharSeq]
             [-metp]
             [-version]
             [-help]
             [-verbosity [1-3]]
             [[-inputIdl] <IDLInputFile.idl> |
       [-inputXml] <XMLInputFile.xml> |
[-inputXsd] <XSDInputFile.xsd> |
[-inputWsdl] <WSDLInputFile.wsdl>]
```

-d Specifies where to put the generated files. If omitted, the input file's directory is used.

-language Generates output for only the language specified. The default is C++.

Use of generated Ada 2005 code requires installation of RTI Ada 2005 Language Support. Please contact `support@rti.com` for more information.

-namespace Specifies the use of C++ namespaces (for C++ only).

-package Specifies a packagePrefix to use as the root package (for Java only).

-example Generates example programs and makefiles (for UNIX-based systems) or workspace and project files (for Windows systems) based on the input types description file.

The <arch> parameter specifies the architecture for the example makefiles.

For -language C/C++, valid options for <arch> are:

sparcSol2.9gcc3.2, sparcSol2.9cc5.4, sparcSol2.10gcc3.4.2, sparc64Sol2.10gcc3.4.2, i86Sol2.9gcc3.3.2, i86Sol2.10gcc3.4.4, x64Sol2.10gcc3.4.3,

x64Darwin10gcc4.2.1,

i86Linux2.6gcc3.4.3, x64Linux2.6gcc3.4.5, i86Linux2.6gcc4.1.1, x64Linux2.6gcc4.1.1, i86Linux2.6gcc4.1.2, x64Linux2.6gcc4.1.2, i86Linux2.6gcc4.2.1, i86Linux2.6gcc4.4.3, x64Linux2.6gcc4.4.3, x64Linux2.6gcc4.3.4, i86Linux2.6gcc4.4.5, x64Linux2.6gcc4.4.5, i86Linux2.6gcc3.4.6, i86RedHawk5.1gcc4.1.2, i86RedHawk5.4gcc4.2.1, x64Linux2.6gcc4.4.4, x64Linux2.6gcc4.5.1, i86Suse10.1gcc4.1.0, x64Suse10.1gcc4.1.0, cell64Linux2.6gcc4.5.1, armv7leLinux2.6gcc4.4.1, i86WRLinux2.6gcc4.3.2, x64WRLinux2.6gcc4.4.1,

ppc4xxFPLinux2.6gcc4.3.3, ppc7400Linux2.6gcc3.3.3, ppc85xxLinux2.6gcc4.3.2, ppc85xxWRLinux2.6gcc4.3.2,

i86Win32VS2005, x64Win64VS2005, i86Win32VS2008, x64Win64VS2008, i86Win32VS2010, x64Win64VS2010,

ppc85xxInty5.0.11.xes-p2020, mpc8349Inty5.0.11.mds8349, pentiumInty10.0.0.pcx86,

ppc7400Lynx4.0.0gcc3.2.2, ppc7400Lynx4.2.0gcc3.2.2, ppc750Lynx4.0.0gcc3.2.2, ppc7400Lynx5.0.0gcc3.4.3, i86Lynx4.0.0gcc3.2.2,

ppc604Vx5.5gcc, ppc603Vx5.5gcc, ppc604Vx6.3gcc3.4.4, ppc604Vx6.3gcc3.4.4_rtp, ppc604Vx6.5gcc3.4.4, ppc604Vx6.5gcc3.4.4_rtp, pentiumVx6.6gcc4.1.2, pentiumVx6.6gcc4.1.2_rtp, ppc405Vx6.6gcc4.1.2, ppc405Vx6.6gcc4.1.2_rtp, ppc604Vx6.6gcc4.1.2, ppc604Vx6.6gcc4.1.2_rtp, pentiumVx6.7gcc4.1.2, pentiumVx6.7gcc4.1.2_rtp, ppc604Vx6.7gcc4.1.2,

ppc604Vx6.7gcc4.1.2_smp, ppc604Vx6.7gcc4.1.2_rtp, ppc604Vx6.8gcc4.1.2, ppc604Vx6.8gcc4.1.2_rtp, pentiumVx6.8gcc4.1.2, pentiumVx6.8gcc4.1.2_rtp, ppc604Vx6.9gcc4.3.3, ppc604Vx6.9gcc4.3.3_rtp, pentiumVx6.9gcc4.3.3, pentiumVx6.9gcc4.3.3_rtp, pentium64Vx6.9gcc4.3.3, pentium64Vx6.9gcc4.3.3_rtp, ppc604VxT2.2.2gcc3.3.2, ppc604VxT2.2.2gcc3.3.2_v6, ppc85xxVxT2.2.3gcc3.3.2, sbc8641Vx653-2.3gcc3.3.2, simpcVx653-2.3gcc3.3.2,

p5AIX5.3xlc9.0, 64p5AIX5.3xlc9.0,

i86QNX6.4.1qcc_gpp i86QNX6.5qcc_gpp4.4.2 i86QNX6.4.0qcc_acpp4.2.4 ppcbeQNX6.4.0qcc_acpp4.2.4

For -language C++/CLI and C#, valid options for <arch> are:

i86Win32dotnet2.0, x64Win64dotnet2.0, i86Win32dotnet4.0, x64Win64dotnet4.0

For -language java, valid options for <arch> are:

i86Sol2.9jdk, i86Sol2.10jdk, x64Sol2.10jdk, sparcSol2.9jdk, sparcSol2.10jdk, sparc64Sol2.10jdk, x64Darwin10gcc4.2.1jdk, i86Linux2.6gcc3.4.3jdk, x64Linux2.6gcc3.4.5jdk, i86Linux2.6gcc4.1.1jdk, x64Linux2.6gcc4.1.1jdk, i86Linux2.6gcc4.4.3jdk, x64Linux2.6gcc4.4.3jdk, i86Linux2.6gcc4.4.5jdk, x64Linux2.6gcc4.4.5jdk, i86Linux2.6gcc4.2.1jdk, x64Linux2.6gcc4.3.4jdk, i86Linux2.6gcc4.1.2jdk, x64Linux2.6gcc4.1.2jdk, i86Linux2.6gcc3.4.6jdk, i86RedHawk5.1gcc4.1.2jdk, i86RedHawk5.4gcc4.2.1jdk, i86Suse10.1gcc4.1.0jdk, x64Suse10.1gcc4.1.0jdk, i86Win32jdk, x64Win64jdk, ppc7400Lynx4.0.0gcc3.2.2jdk, ppc750Lynx4.0.0gcc3.2.2jdk, ppc7400Lynx5.0.0gcc3.4.3jdk, i86Lynx4.0.0gcc3.2.2jdk, p5AIX5.3xlc9.0jdk, 64p5AIX5.3xlc9.0jdk

For -language Ada, valid option for <arch> is i86Linux2.6gcc4.1.2

-replace Overwrites any existing output files. Warning: This removes any changes you may have made to the original files.

-debug Generates intermediate files for debugging purposes.

-corba [client header file] [-orb <CORBA ORB>] Specifies that you want to produce CORBA-compliant code.

Use [client header file] and [-orb <CORBA ORB>] for C++ only. The majority of code generated is independent of the ORB. However, for some IDL features, the code generated depends on the ORB. This version of rtiddsgen generates code compatible with ACE-TAO or JacORB. To pick the ACE_TAO version, use the -orb parameter; the default is ACE_TAO1.6.

client header file: the name of the header file for the IDL types generated by the CORBA IDL compiler. This file will be included in the rtiddsgen type header file instead of generating type definitions.

CORBA support requires the RTI CORBA Compatibility Kit, an add-on product that provides a different version of rtiddsgen. Please contact `support@rti.com` for more information.

-optimization Sets the optimization level. (Only applies to C/C++)

- 0 (default): No optimization.
- 1: Compiler generates extra code for typedefs but optimizes its use. If the type that is used is a typedef that can be resolved either to a primitive type or to another type defined in the same file, the generated code will invoke the code of the most basic type to which the typedef can be resolved, unless the most basic type is an array or a sequence. This level can be used if the generated code is not expected to be modified.
- 2: Maximum optimization. Functionally the same as level 1, but extra code for typedef is not generated. This level can be used if the typedefs are only referred by types within the same file.

-typeSequenceSuffix Assigns a suffix to the name of the implicit sequence defined for IDL types. (Only applies to CORBA)

By default, the suffix is 'Seq'. For example, given the type 'Foo' the name of the implicit sequence will be 'FooSeq'.

-dataReaderSuffix Assigns a suffix to the name of the DataReader interface. (Only applies to CORBA)

By default, the suffix is 'DataReader'. For example, given the type 'Foo' the name of the DataReader interface will be 'FooDataReader'.

-dataWriterSuffix Assigns a suffix to the name of the DataWriter interface. (Only applies to CORBA)

By default, the suffix is 'DataWriter'. For example, given the type 'Foo' the name of the DataWriter interface will be 'FooDataWriter'.

-stringSize Sets the size for unbounded strings. Default: 255 bytes.

-sequenceSize Sets the size for unbounded sequences. Default: 100 elements.

-notypecode: Disables the generation of type code information.

-ppDisable: Disables the preprocessor.

-ppPath <preprocessor executable>: Specifies the preprocessor path. If you only specify the name of an executable (not a complete path to that executable), the executable must be found in your Path.

The default value is "cpp" for non-Windows architectures, "cl.exe" for Windows architectures.

If the default preprocessor is not found in your Path and you use -ppPath to provide its full path and filename, you must also use -ppOption (described

below) to set the following preprocessor options:

- If you use a non-default path for cl.exe, you also need to set:
  -ppOption /nologo -ppOption /C -ppOption /E -ppOption /X

- If you use a non-default path for cpp, you also need to set:
  -ppOption -C

-ppOption <option>: Specifies a preprocessor option. This parameter can be used multiple times to provide the command-line options for the specified preprocessor. See -ppPath (above).

-D <name>[=<value>]: Defines preprocessor macros.

-U <name>: Cancels any previous definition of name.

-I <directory>: Adds to the list of directories to be searched for type-definition files (IDL, XML, XSD or WSDL files). Note: A type-definition file in one format cannot include a file in another format.

-noCopyable: Forces rtiddsgen to put copy logic into the corresponding TypeSupport class rather than the type itself (for Java code generation only).

This option is not compatible with the use of ndds_standalone_type.jar.

-use42eAlignment: Generates code compliant with RTI Data Distribution Service 4.2e.

If your RTI Connext application's data type uses a 'double','long long','unsigned long long', or 'long double' it will not be backwards compatible with RTI Data Distribution Service 4.2e applications unless you use the -use42eAlignment flag when generating code with rtiddsgen.

-enableEscapeChar: Enables use of the escape character '_' in IDL identifiers. With CORBA this option is always enabled.

-convertToXml: Converts the input type-description file to XML format.

-convertToIdl: Converts the input type-description file to IDL format.

-convertToXsd: Converts the input type-description file to XSD format.

-convertToWsdl: Converts the input type-description file to WSDL format.

-convertToCcl: Converts the input type-description file to CCL format.

-convertToCcs: Converts the input type-description file to CCS format.

-expandOctetSeq: When converting to CCS or CCL files, expand octet sequences. The default is to use a blob type.

-expandCharSeq: When converting to CCS or CCL files, expand char sequences. The default is to use a string type.

-metp: Generates code for the Multi-Encapsulation Type Support (METP) library.

-version: Prints the version, such as 4.5x. (Does not show 'patch' revision number.)

-help: Prints this rtiddsgen usage help.

-verbosity: rtiddsgen verbosity.

- 1: exceptions
- 2: exceptions and warnings
- 3 (default): exceptions, warnings and information

-inputIdl: Indicates that the input file is an IDL file, regardless of the file extension.

-inputXml: Indicates that the input file is a XML file, regardless of the file extension.

-inputXsd: Indicates that the input file is a XSD file, regardless of the file extension.

-inputWsdl: Indicates that the input file is a WSDL file, regardless of the file extension.

IDLInputFile.idl: File containing IDL descriptions of your data types. If -inputIdl is not used, the file must have an .idl extension.

XMLInputFile.xml: File containing XML descriptions of your data types. If -inputXml is not used, the file must have an .xml extension.

XSDInputFile.xsd: File containing XSD descriptions of your data types. If -inputXsd is not used, the file must have an .xsd extension.

XSDInputFile.wsdl: WSDL file containing XSD descriptions of your data types. If -inputWsdl is not used, the file must have an .wsdl extension.

5.59.2 Description

rtiddsgen takes a language-independent specification of the data (in IDL, XML, XSD or WSDL notation) and generates supporting classes and code to distribute instances of the data over RTI Connext.

To use rtiddsgen, you must first write a description of your data types in IDL, XML, XSD or WSDL format.

5.59.3 C++ Example

The following is an example generating the RTI Connext type myDataType:

IDL notation

```
struct myDataType {
    long value;
};
```

XML notation

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="rti_dds_topic_types.xsd">
    <struct name="myDataType">
        <member name="value" type="long"/>
    </struct>
</types>
```

XSD notation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:dds="http://www.omg.org/dds"
     xmlns:tns="http://www.omg.org/IDL-Mapped/"
     targetNamespace="http://www.omg.org/IDL-Mapped/">
  <xsd:import namespace="http://www.omg.org/dds" schemaLocation="rti_dds_topic_types_common.xsd"/
  <xsd:complexType name="myDataType">
    <xsd:sequence>
      <xsd:element name="value" minOccurs="1" maxOccurs="1" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

WSDL notation

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:dds="http://www.omg.org/dds" xmlns:x
  <types>
    <xsd:schema targetNamespace="http://www.omg.org/IDL-Mapped/">
      <xsd:import namespace="http://www.omg.org/dds" schemaLocation="rti_dds_topic_types_common.x
      <xsd:complexType name="myDataType">
        <xsd:sequence>
          <xsd:element name="value" minOccurs="1" maxOccurs="1" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
```

```
    </xsd:schema>
  </types>
</definitions>
```

Assuming the name of the idl file is myFileName.(idl|xml|xsd|wsdl) then all you need to do is type:

rtiddsgen myFileName.(idl|xml|xsd|wsdl)

This generates myFileName.cxx, myFileName.h, myFileNamePlugin.cxx, myFileNamePlugin.h, myFileNameSupport.cxx and myFileNameSupport.h. By default, rtiddsgen will not overwrite these files. You must use the -replace argument to do that.

5.59.4 IDL Language

In the IDL language, data types are described in a fashion almost identical to structures in "C." The complete description of the language can be found at the OMG website.

rtiddsgen does not support the full IDL language.

For detailed information about the IDL support in RTI Connext see Chapter 3 in the `User's Manual`.

The supported IDL types are:

- char
- wchar
- octet
- short
- unsigned short
- long
- unsigned long
- long long
- unsigned long long
- float
- double
- long double
- boolean

- bounded string
- unbounded string
- bounded wstring
- unbounded wstring
- enum
- typedef
- struct
- valuetypes (limited support)
- union
- sequences
- unbounded sequences
- arrays
- array of sequences
- constant

The following non-IDL types are also supported:

- bitfield
- valued enum

Use of Unsupported Types in an IDL File

You may include unsupported data types in the IDL file. rtiddsgen does not consider this an error. This allows you to use types that are defined in non-IDL languages with either hand-written or non-rtiddsgen written plug-ins. For example, the following is allowable:

```
//@copy #include "Bar.h"
//@copy #include "BarHandGeneratedPlugin.h"
struct Foo {
short height;
Bar barMember;
};
```

In the above case, Bar is defined externally by the user.

Multiple Types in a Single File

You can specify multiple types in a single IDL file. This can simplify management of files in your distributed program.

Use of Directives in an IDL File

The following directives can be used in your IDL file: Note: Do not put a space between the slashes and the @ sign. Note: Directives are case-sensitive (for example: use key, not Key).

- //@key Specifies that the field declared just before this directive in the enclosing structure is part of the key. Any number of a structure's fields may be declared part of the key.
- //@copy Copies a line of text (verbatim) into the generated code (for all languages). The text is copied into all the type-specific files generated by rtiddsgen except the examples.
- //@copy-declaration Like //@copy, but only copies the text into the file where the type is declared (<type>.h for C++/C, or <type>.java for Java).
- //@copy-c Like //@copy, but for C++/C-only code.
- //@copy-c-declaration Like //@copy-declaration, but for C++/C-only code.
- //@copy-java Like //@copy, but for Java-only code.
- //@copy-java-begin Copies a line of text at the beginning of all the Java files generated for a type. The directive only applies to the first type that is immediately below in the IDL file.
- //@copy-java-declaration Like //@copy-declaration, but for Java-only code.
- //@copy-java-declaration-begin Like //@copy-java-begin but only copies the text into the file where the type is declared.
- //@copy-ada Like //@copy, but for Ada-only code.
- //@copy-ada-begin Like //@copy-java-begin, but for Ada-only code.
- //@copy-ada-declaration Like //@copy-declaration, but for Ada-only code.
- //@copy-ada-declaration-begin Like //@copy-java-declaration, but for Ada-only code.

- //@resolve-name [true|false] Specifies whether or not rtiddsgen should resolve the scope of a type. If this directive is not present or is set to true, rtiddsgen resolves the scope. Otherwise rtiddsgen delegates the resolution of a type to the user.

- //@top-level [true|false] Specifies whether or not rtiddsgen should generate type-support code for a particular struct or union. The default is true.

5.59.5 XML Language

The data types can be described using XML.

RTI Connext provides DTD and XSD files that describe the XML format.

The DTD definition of the XML elements can be found in ../../../resource/dtd/rti_dds_topic_types.dtd" under <NDDSHOME>/resource/rtiddsgen/schema.

The XSD definition of the XML elements can be found in ../../../resource/xsd/rti_dds_topic_types.xsd" under <NDDSHOME>/resource/rtiddsgen/schema.

The XML validation performed by rtiddsgen always uses the DTD definition. If the <!DOCTYPE> tag is not present in the XML file, rtiddsgen will look for the DTD document under <NDDSHOME>/resource/rtiddsgen/schema. Otherwise, it will use the location specified in <!DOCTYPE>.

For detailed information about the mapping between IDL and XML, see Chapter 3 in the `User's Manual`.

5.59.6 XSD Language

The data types can be described using XML schemas (XSD files). The XSD specification is based on the standard IDL to WSDL mapping described in the OMG document `CORBA to WSDL/SOAP Interworking Specification`

For detailed information about the mapping between IDL and XML see Chapter 3 in the `User's Manual`.

5.59.7 WSDL Language

The data types can be described using XML schemas contained in WSDL files. The XSD specification is based on the standard IDL to WSDL mapping described in the OMG document `CORBA to WSDL/SOAP Interworking Specification`

For detailed information about the mapping between IDL and XML see Chapter 3 in the `User's Manual`.

5.59.8 Using Generated Types Without RTI Connext (Standalone)

You can use the generated type-specific source and header files without linking the RTI Connext libraries or even including the RTI Connext header files. That is, the generated files for your data types can be used standalone.

The directory <NDDSHOME>/resource/rtiddsgen/standalone contains the helper files required to work in standalone mode:

- include: header and templates files for C/C++.
- src: source files for C/C++.
- class: Java jar file.

Using Standalone Types in C

The generated files that can be used standalone are:

- <idl file name>.c : Types source file
- <idl file name>.h : Types header file

You *cannot* use the type plug-in (<idl file>Plugin.c <idl file>Plugin.h) or the type support (<idl file>Support.c <idl file>Support.h) code standalone.

To use the rtiddsgen-generated types in a standalone manner:

- Include the directory <NDDSHOME>/resource/rtiddsgen/standalone/include in the list of directories to be searched for header files.
- Add the source files ndds_standalone_type.c and <idl file name>.c to your project.
- Include the file <idl file name>.h in the source files that will use the generated types in a standalone way.
- Compile the project using the two following preprocessor definitions:
  - NDDS_STANDALONE_TYPE
  - The definition for your platform: RTI_VXWORKS, RTI_QNX, RTI_WIN32, RTI_INTY, RTI_LYNX or RTI_UNIX

Using Standalone Types in C++

The generated files that can be used standalone are:

- <idl file name>.cxx : Types source file
- <idl file name>.h : Types header file

You *cannot* use the type plugin (<idl file>Plugin.cxx <idl file>Plugin.h) or the type support (<idl file>Support.cxx <idl file>Support.h) code standalone.

To use the generated types in a standalone manner:

- Include the directory <NDDSHOME>/resource/rtiddsgen/standalone/include in the list of directories to be searched for header files.
- Add the source files ndds_standalone_type.cxx and <idl file name>.cxx to your project.
- Include the file <idl file name>.h in the source files that will use the generated types in a standalone way.
- Compile the project using the two following preprocessor definitions:
  - NDDS_STANDALONE_TYPE
  - The definition for your platform: RTI_VXWORKS, RTI_QNX, RTI_WIN32, RTI_INTY, RTI_LYNX or RTI_UNIX

Standalone Types in Java

The generated files that can be used standalone are:

- <idl type>.java
- <idl type>Seq.java

You *cannot* use the type code (<idl file>TypeCode.java), the type support (<idl type>TypeSupport.java), the data reader (<idl file>DataReader.java) or the data writer code (<idl file>DataWriter.java) standalone.

To use the generated types in a standalone manner:

- Include the file ndds_standalone_type.jar in the classpath of your project.
- Compile the project using the standalone types files (<idl type>.java <idl type>Seq.java).

5.60 rtiddsping

Sends or receives simple messages using RTI Connext. The rtiddsping utility uses RTI Connext to send and receive preconfigured "Ping" messages to other rtiddsping applications which can be running in the same or different computers.

The rtiddsping utility can used to test the network and/or computer configuration and the environment settings that affect the operation of RTI Connext.

Usage

```
  rtiddsping [-help] [-version]
    [-domainId <domainId>]     ... defaults to 0
    [-index <NN>]              ... defaults to -1 (auto)
    [-appId <ID>]              ... defaults to a middleware-selected value
    [-Verbosity <NN>]          ... can be 0..5
    [-peer <PEER>]             ... PEER format is NN@TRANSPORT://ADDRESS
    [-discoveryTTL <NN>]       ... can be 0..255
[-transport <MASK>]        ... defaults to DDS_TRANSPORTBUILTIN_MASK_DEFAULT
[-msgMaxSize <SIZE>]       ... defaults to -1 (transport default)
[-shmRcvSize <SIZE>]       ... defaults to -1 (transport default)
[-deadline   <SS>]         ... defaults to -1 (no deadline)
[-durability <TYPE>]       ... TYPE can be VOLATILE or TRANSIENT_LOCAL
    [-multicast  <ADDRESS>]    ... defaults to no multicast
[-numSamples <NN>]         ... defaults to infinite
    [-publisher]                 ... this is the default
[-queueSize  <NN>]         ... defaults to 1
[-reliable]                  ... defaults to best-efforts
[-sendPeriod <SS>]         ... SS is in seconds, defaults to 1
[-subscriber]
[-timeFilter <SS>]         ... defaults to  0 (no filter)
[-timeout    <SS>]         ... SS is in seconds, defaults to infinite
[-topicName  <NAME>]       ... defaults to PingTopic
[-typeName   <NAME>]       ... defaults to PingType
[-useKeys    <NN>]         ... defaults to PingType
    [-qosFile <file>]
    [-qosProfile  <lib::prof>]
```

Example: `rtiddsping -domainId 3 -publisher -numSamples 100`

VxWorks Usage

rtiddsping `"[<options>]"`

The options use the same syntax as above.

Example `rtiddsping "-domainId 3 -publisher -numSamples 100"`

If the stack of the shell is not large enough to run rtiddsping, use "taskSpawn":

taskSpawn <name>,<priority>,<taskspawn options>,<stack size in bytes>,rtiddsping,"[\<options
The options use the same syntax as above.

Example taskSpawn "rtiddsping",100,0x8,50000,rtiddsping,"-domainId 3 -publisher -numSamples

Options:

-help Prints a help message and exits.

-version Prints the version and exits.

-Verbosity <NN> Sets the verbosity level. The range is 0 to 5.

0 has minimal output and does not echo the fact that data is being sent or received.

1 prints the most relevant statuses, including the sending and receiving of data. This is the default.

2 prints a summary of the parameters that are in use and echoes more detailed status messages.

3-5 Mostly affects the verbosity used by the internal RTI Connext modules that implement rtiddsping. The output is not always readable; its main purpose is to provide information that may be useful to RTI's support team.

Example: rtiddsping -Verbosity 2

-domainId <NN>

Sets the domain ID. The valid range is 0 to 100.

Example: rtiddsping -domainId 31

-appId <ID>

Sets the application ID. If unspecified, the system will pick one automatically.

This option is rarely used.

Example: rtiddsping -appId 34556

-index <NN>

Sets the participantIndex. If participantIndex is not -1 (auto), it must be different than the one used by all other applications in the same computer and domainId. If this is not respected, rtiddsping (or the application that starts last) will get an initialization error.

Example: rtiddsping -index 2

-peer <PEER>

Specifies a PEER to be used for discovery. Like any RTI Connext application, it defaults to the setting of the environment variable NDDS_DISCOVERY_PEERS

or a preconfigured multicast address if the environment is not set.

The format used for PEER is the same one used for NDDS_DISCOVERY_PEERS and is described in detail in **NDDS_DISCOVERY_PEERS** (p. 388). A brief summary follows:

The general format is: NN@TRANSPORT://ADDRESS where:

- ADDRESS is an address (in name form or using the IP notation xxx.xxx.xxx.xxx). ADDRESS may be a multicast address.
- TRANSPORT represents the kind of transport to use and NN is the maximum participantIndex expected at that location. NN can be omitted and it is defaulted to '4'
- Valid settings for TRANSPORT are 'udpv4' and 'shmem'. The default setting if the transport is omitted is 'udpv4'.
- ADDRESS cannot be omitted if the '-peer' option is specified.

The -peer option may be repeated to specify multiple peers.

Example: rtiddsping -peer 10.10.1.192 -peer mars -peer 4@pluto

-discoveryTTL <TTL>

Sets the TTL (time-to-live) used for multicast discovery. If not specified, it defaults to the built-in RTI Connext default.

The valid range is 0 to 255. The value '0' limits multicast to the node itself (i.e., can only discover applications running on the same computer). The value '1' limits multicast discovery to computers on the same subnet. Values higher than 1 generally indicate the maximum number of routers that may be traversed (although some routers may be configured differently).

Example: rtiddsping -discoveryTTL 16

-transport <MASK>

A bit-mask that sets the enabled builtin transports. If not specified, the default set of transports is used (UDPv4 + shmem). The bit values are: 1=UDPv4, 2=shmem, 8=UDPv6.

-msgMaxSize <SIZE>

Configure the maximum message size allowed by the installed transports. This is needed if you are using rtiddsping to communicate with an application that has set these transport parameters to larger than default values.

-shmRcvSize <SIZE>

Increase the shared memory receive-buffer size. This is needed if you are using rtiddsping to communicate with an application that has set these transport parameters to larger than default values.

-deadline <SS>

This option only applies if the '-subscriber' option is also specified.

Sets the DEADLINE QoS for the subscriptions made by rtiddsping.

Note that this may cause the subscription QoS to be incompatible with the publisher if the publisher did not specify a sendPeriod greater than the deadline. If the QoS is incompatible, rtiddsping will not receive updates.

Each time a deadline is detected, rtiddsping will print a message indicating the number of deadlines received so far.

Example: rtiddsping -deadline 3.5

-durability <TYPE>

Sets the DURABILITY QoS used for publishing or subscribing. Valid settings are: VOLATILE and TRANSIENT_LOCAL (default). The effect of this setting can only be observed when it is used in in conjunction with reliability and a queueSize larger than 1. If all these conditions are met, a late-joining subscriber will be able to see up to queueSize samples that were previously written by the publisher.

Example: rtiddsping -durability VOLATILE

-multicast <ADDRESS>

This option only applies if the '-subscriber' option is also specified.

Configures ping to receive messages over multicast. The <ADDRESS> parameter indicates the address to use. ADDRESS must be in the valid range for multicast addresses. For IP version 4 the valid range is 224.0.0.1 to 239.255.255.255

Example: rtiddsping -multicast 225.1.1.1

-numSamples <NN>

Sets the number of samples that will be sent by rtiddsping. After those samples are sent, rtiddsping will exit. messages.

Example: rtiddsping -numSamples 10

-publisher

Causes rtiddsping to send ping messages. This is the default.

Example: rtiddsping -publisher

-queueSize <NN>

Specifies the maximal number of samples to hold in the queue. In the case of the publisher, it affects the samples that are available for a late-joining subscriber.

Example: rtiddsping -queueSize 100

-reliable

Configures the RELIABILITY QoS for publishing or subscribing. The default setting (if -reliable is not used) is BEST_EFFORT

Example: rtiddsping -reliable

-sendPeriod <SS>

Sets the period (in seconds) at which rtiddsping sends the messages.

Example: rtiddsping -sendPeriod 0.5

-subscriber

Causes rtiddsping to listen for ping messages. This option cannot be specified if '-publisher' is also specified.

Example: rtiddsping -subscriber

-timeFilter <SS>

This option only applies if the '-subscriber' option is also specified.

Sets the TIME_BASED_FILTER QoS for the subscriptions made by rtiddsping. This QoS causes RTI Connext to filter out messages that are published at a rate faster than what the filter duration permits. For example, if the filter duration is 10 seconds, messages will be printed no faster than once every 10 seconds.

Example: rtiddsping -timeFilter 5.5

-timeout <SS>

This option only applies if the '-subscriber' option is also specified.

Sets a timeout (in seconds) that will cause rtiddsping to exit if no samples are received for a duration that exceeds the timeout.

Example: rtiddsping -timeout 30

-topicName <NAME>

Sets the topic name used by rtiddsping. The default is 'RTIddsPingTopic'. To communicate, both the publisher and subscriber must specify the same topic name.

Example: rtiddsping -topicName Alarm

-typeName <NAME>

Sets the type name used by rtiddsping. The default is 'RTIddsPingType'. To communicate, both publisher and subscriber must specify the same type name.

Example: rtiddsping -typeName AlarmDescription

-useKeys <NN>

This option causes rtiddsping to use a topic whose data contains a key. The value of the NN parameter indicates the number of different data objects (each identified by a different value of the key) that will be published by rtiddsping.

The value of NN only affects the publishing behavior. However NN still needs to be specified when the -useKeys option is used with the -subscriber option.

For communication to occur, both the publisher and subscriber must agree on whether the topic that they publish/subscribe contains a key. Consequently, if you specify the -useKeys parameter for the publisher, you must do the same with the subscriber. Otherwise communication will not be stablished.

Example: rtiddsping -useKeys 20

-qosFile <file>

Allow you to specify additional QoS XML settings using url_profile. For more information on the syntax, see Chapter 15 in the RTI Connext User's Manual.

Example: rtiddsping -qosFile /home/user/QoSProfileFile.xml

-qosProfile <lib::prof>

This option specifies the library name and profile name that the tool should use.

QoS settings

rtiddsping is configured internally using a special set of QoS settings in a profile called InternalPingLibrary::InternalPingProfile. This is the default profile unless a profile called DefaultPingLibrary::DefaultPingProfile is found. You can use the command-line option -qosProfile to tell rtiddsping to use a different lib::profile instead of DefaultPingLibrary::DefaultPingProfile. Like all the other RTI Connext applications, rtiddsping loads all the profiles specified using the environment variable NDDS_QOS_PROFILES or the file named USER_QOS_PROFILES found in the current working directory.

The QoS settings used internally are available in the file RTIDDSSPING_QOS_PROFILES.example.xml.

Description

The usage depends on the operating system from which rtiddsping is executed.

Examples for UNIX, Linux, and Windows Systems

On UNIX, Linux, Windows and other operating systems that have a shell, the syntax matches the one of the regular commands available in the shell. In the examples below, the string 'shell prompt>' represents the prompt that the shell prints and are not part of the command that must be typed.

```
shell prompt>  rtiddsping -domainId 3 -publisher -numSamples 100
shell prompt>  rtiddsping -domainId 5 -subscriber -timeout 20
shell prompt>  rtiddsping -help
```

VxWorks examples:

On VxWorks systems, the libraries libnddscore.so, libnddsc.so and libnddscpp.so must first be loaded. The rtiddsping command must be typed to the VxWorks

shell (either an rlogin shell, a target-server shell, or the serial line prompt). The arguments are passed embedded into a single string, but otherwise have the same syntax as for Unix/Windows. In the Unix, Linux, Windows and other operating systems that have a shell, the syntax matches the one of the regular commands available in the shell. In the examples below, the string 'vxworks prompt>' represents the prompt that the shell prints and are not part of the command that must be typed.

```
vxworks prompt>  rtiddsping "-domainId 3 -publisher -numSamples 100"
vxworks prompt>  rtiddsping "-domainId 5 -subscriber -timeout 20"
vxworks prompt>  rtiddsping "-help"

or, alternatively (to avoid overflowing the stack):

vxworks prompt> taskSpawn "rtiddsping", 100, 0x8, 50000, rtiddsping, "-domainId 3 -publisher -numSamples 10
vxworks prompt> taskSpawn "rtiddsping", 100, 0x8, 50000, rtiddsping, "-domainId 5 -subscriber -timeout 20"
vxworks prompt> taskSpawn "rtiddsping", 100, 0x8, 50000, rtiddsping, "-help"
```

5.61 rtiddsspy

Debugging tool which receives all RTI Connext communication. The rtiddsspy utility allows the user to monitor groups of publications available on any RTI Connext domain.

Note: If you have more than one DataWriter for the same Topic, and these DataWriters have different settings for the Ownership QoS, then rtiddsspy will only receive (and thus report on) the samples from the first DataWriter.

To run rtiddsspy, like any RTI Connext application, you must have the NDDS_DISCOVERY_PEERS environment variable that defines your RTI Connext domain; otherwise you must specify the peers as command line parameters.

Usage

```
rtiddsspy [-help] [-version]
     [-domainId <domainId>]     ... defaults to 0
     [-index <NN>]              ... defaults to -1 (auto)
     [-appId <ID>]              ... defaults to a middleware-selected value
     [-Verbosity <NN>]          ... can be 0..5
     [-peer <PEER>]             ... PEER format is NN@TRANSPORT://ADDRESS
     [-discoveryTTL <NN>]       ... can be 0..255
     [-transport <MASK>]        ... defaults to DDS_TRANSPORTBUILTIN_MASK_DEFAULT
     [-msgMaxSize <SIZE>]       ... defaults to -1 (transport default)
     [-shmRcvSize <SIZE>]       ... defaults to -1 (transport default)
     [-tcMaxSize <SIZE>]        ... defaults to 4096
     [-hOutput]
     [-deadline   <SS>]         ... defaults to -1 (no deadline)
     [-history   <DEPTH>]       ... defaults to 8192
     [-timeFilter <SS>]         ... defaults to  0 (no filter)
     [-useFirstPublicationQos]
     [-showHandle]
     [-typeRegex <REGEX>]       ... defaults to "*"
     [-topicRegex <REGEX>]      ... defaults to "*"
     [-typeWidth <WIDTH>]       ... can be 1..255
     [-topicWidth <WIDTH>]      ... can be 1..255
     [-truncate]
     [-printSample]
     [-qosFile <file>]
     [-qosProfile  <lib::prof>]
```

Example: `rtiddsspy -domainId 3 -topicRegex "Alarm*"`

VxWorks Usage

```
rtiddsspy "[<options>]"
     The options use the same syntax as above.
```

Example `rtiddsspy "-domainId 3 -topicRegex Alarm*"`

`rtiddsspy requires about 25 kB of stack. If the stack size of the shell from which it is invoked is not la`

taskSpawn `<name>, <priority>, <taskspawn options>, <stack size in bytes>, rtiddsspy, "[\<options\>]"`

`The options use the same syntax as above.`

Example `taskSpawn "rtiddsspy", 100, 0x8, 50000, rtiddsspy, "-domainId 3 -topicRegex Alarm*"`

Options:

-help Prints a help message and exits.

-version Prints the version and exits.

-Verbosity <NN> Sets the verbosity level. The range is 0 to 5.

0 has minimal output and does not echo the fact that data is being sent or received.

1 prints the most relevant statuses, including the sending and receiving of data. This is the default.

2 prints a summary of the parameters being used and echoes more detailed status messages.

3-5 Mostly affect the verbosity used by the internal RTI Connext modules that implement rtiddsspy. The output is not always readable; its main purpose is to provide information that may be useful to RTI's support team.

Example: rtiddsspy -Verbosity 2

-domainId <NN>

Sets the domain ID. The valid range is 0 to 100.

Example: rtiddsspy -domainId 31

-appId <ID>

Sets the application ID. If unspecified, the system will pick one automatically.

This option is rarely used.

Example: rtiddsspy -appId 34556

-index <NN>

Sets the participantIndex. If participantIndex is not -1 (auto), it must be different than the one used by all other applications in the same computer and domainId. If this is not respected, rtiddsspy (or the application that starts last) will get an initialization error.

Example: rtiddsspy -index 2

-peer <PEER>

Specifies a PEER to be used for discovery. Like any RTI Connext application, it defaults to the setting of the environment variable NDDS_DISCOVERY_PEERS or a preconfigured multicast address if the environment is not set.

The format used for PEER is the same used for the NDDS_DISCOVERY_PEERS and is described in detail in **NDDS_DISCOVERY_PEERS** (p. 388). A brief summary follows:

The general format is: NN@TRANSPORT://ADDRESS where:

- ADDRESS is an address (in name form or using the IP notation xxx.xxx.xxx.xxx). ADDRESS may be a multicast address.
- TRANSPORT represents the kind of transport to use and NN is the maximum participantIndex expected at that location. NN can be omitted and it is defaulted to '4'
- Valid settings for TRANSPORT are 'udpv4' and 'shmem'. The default setting if the transport is omitted is 'udpv4'
- ADDRESS cannot be omitted if the '-peer' option is specified.

The -peer option may be repeated to specify multiple peers.

Example: rtiddsspy -peer 10.10.1.192 -peer mars -peer 4@pluto

-discoveryTTL <TTL>

Sets the TTL (time-to-live) used for multicast discovery. If not specified, it defaults to the built-in RTI Connext default.

The valid range is 0 to 255. The value '0' limits multicast to the node itself (i.e. can only discover applications running on the same computer). The value '1' limits multicast discovery to computers on the same subnet. Settings greater than 1 generally indicate the maximum number of routers that may be traversed (although some routers may be configured differently).

Example: rtiddsspy -discoveryTTL 16

-transport <MASK>

SPecifies a bit-mask that sets the enabled builtin transports. If not specified, the default set of transports is used (UDPv4 + shmem). The bit values are: 1=UDPv4, 2=shmem, 8=UDPv6.

-msgMaxSize <SIZE>

Configures the maximum message size allowed by the installed transports. This is needed if you are using rtiddsspy to communicate with an application that has set these transport parameters to larger than default values.

-shmRcvSize <SIZE>

Increases the shared memory receive-buffer size. This is needed if you are using rtiddsspy to communicate with an application that has set these transport parameters to larger than default values.

-tcMaxSize <SIZE>

Configures the maximum size, in bytes, of a received type code.

-hOutput

Prints information on the output format used by rtiddsspy.

This option prints an explanation of the output and then exits.

Example: rtiddsspy -hOutput

-deadline <SS>

Sets the requested DEADLINE QoS for the subscriptions made by rtiddsspy.

Note that this may cause the subscription QoS to be incompatible with the publisher if the publisher did not specify an offered deadline that is greater or equal to the one requested by rtiddsspy. If the QoS is incompatible rtiddsspy will not receive updates from that writer.

Each time a deadline is detected rtiddsspy will print a message that indicates the number of deadlines received so far.

Example: rtiddsspy -deadline 3.5

-timeFilter <SS>

Sets the TIME_BASED_FILTER QoS for the subscriptions made by rtiddsspy. This QoS causes RTI Connext to filter-out messages that are published at a rate faster than what the filter duration permits. For example if the filter duration is 10 seconds, messages will be printed no faster than once each 10 seconds.

Example: rtiddsspy -timeFilter 10.0

-history <DEPTH>

Sets the HISTORY depth QoS for the subscriptions made by rtiddsspy.

This may be relevant if the publisher has batching turned on, or if the -useFirstPublicationQos option is used that is causing a reliable or durable subscription to be created.

Example: rtiddsspy -history 1

-useFirstPublicationQos

Sets the RELIABILITY and DURABILITY QoS of the subscription based on the first discovered publication of that topic.

See also -history option.

Example: rtiddsspy -useFirstPublicationQos

-showHandle

Prints additional information about each sample received. The additional information is the 'instance_handle' field in the SampleHeader, which can be used to distinguish among multiple instances of data objects published under the same topic and type names.

Samples displayed that share the topic and type names and also have the same value for the instance_handle represent value updates to the same data object. On the other hand, samples that share the topic and type names but display different values for the instance_handle.

This option causes rtiddsspy to print an explanation of updates to the values of different data objects.

Example: rtiddsspy -showHandle

-typeRegex <REGEX>

Subscribe only to types that match the REGEX regular expression. The syntax of the regular expression is defined by the POSIX regex function.

When typing a regular expression to a command-line shell, some symbols may need to be escaped to avoid interpretation by the shell. In general, it is safest to include the expression in double quotes.

This option may be repeated to specify multiple topic expressions.

Example: rtiddsspy -typeRegex "SensorArray*"

-topicRegex <REGEX>

Subscribe only to topics that match the REGEX regular expression. The syntax of the regular expression is defined by the POSIX regex function.

When typing a regular expression to a command-line shell, some symbols may need to be escaped to avoid interpretation by the shell. In general, it is safest to include the expression in double quotes.

This option may be repeated to specify topic multiple expressions.

Example: rtiddsspy -topicRegex "Alarm*"

-typeWidth <WIDTH>

Sets the maximum width of the Type name column. Names wider than this will wrap around, unless `-truncate` is specified. Can be 1..255.

-topicWidth <WIDTH>

Sets the maximum width of the Topic name column. Names wider than this will wrap around, unless `-truncate` is specified. Can be 1..255.

-truncate

Specifies that names exceeding the maximum number of characters should be truncated.

-printSample

Prints the value of the received samples.

-qosFile <file>

Allows you to specify additional QoS XML settings using url_profile. For more information on the syntax, see Chapter 15 in the RTI Connext User's Manual.

Example: rtiddsspy -qosFile /home/user/QoSProfileFile.xml

-qosProfile <lib::prof>

Specifies the library name and profile name to be used.

QoS settings

rtiddsspy is configured to discover as many entities as possible. To do so, an internal profile is defined, called InternalSpyLibrary::InternalSpyProfile. This is the default profile, unless a profile called DefaultSpyLibrary::DefaultSpyProfile is found. You can use the command-line option -qosProfile to tell rtiddsspy to use a speficied lib::profile instead of DefaultSpyLibrary::DefaultSpyProfile. Like all the other RTI Connext applications, rtiddsspy loads all the profiles specified using the environment variable NDDS_QOS_PROFILES or the file named USER_QOS_PROFILES found in the current working directory.

The QoS settings used internally are available in the file RTIDDSSPY_QOS_PROFILES.example.xml.

Usage Examples

The usage depends on the operating system from which rtiddsspy is executed.

Examples for UNIX, Linux, Windows systems

On UNIX, Linux, Windows and other operating systems that have a shell, the syntax matches the one of the regular commands available in the shell. In the examples below, the string 'shell prompt>' represents the prompt that the shell prints and are not part of the command that must be typed.

```
shell prompt>  rtiddsspy -domainId 3
shell prompt>  rtiddsspy -domainId 5 -topicRegex "Alarm*"
shell prompt>  rtiddsspy -help
```

Examples for VxWorks Systems

On VxWorks systems, the libraries libnddscore.so, libnddsc.so and libnddscpp.so must first be loaded. The rtiddsspy command must be typed to the VxWorks shell (either an rlogin shell, a target-server shell, or the serial line prompt). The arguments are passed embedded into a single string, but otherwise have the same syntax as for Unix/Windows. In UNIX, Linux, Windows and other

operating systems that have a shell, the syntax matches the one of the regular comamnds available in the shell. In the examples below, the string 'vxworks prompt>' represents the prompt that the shell prints and are not part of the command that must be typed.

```
vxworks prompt>  rtiddsspy "-domainId 3"
vxworks prompt>  rtiddsspy "-domainId 5  5 -topicRegex "Alarm*"
vxworks prompt>  rtiddsspy "-help"
```

5.62 Class Id

Transport class ID. Specifies the ID of a Transport-Plugin 'class'.

Defines

- #define **NDDS_TRANSPORT_CLASSID_INVALID** (-1)

  Invalid Transport Class ID.

- #define **NDDS_TRANSPORT_CLASSID_DTLS** (6)

  DTLS Secure Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_CLASSID_WAN** (7)

  WAN Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_CLASSID_TCPV4_LAN** (8)

  IPv4 TCP/IP Transport-Plugin class ID for LAN case.

- #define **NDDS_TRANSPORT_CLASSID_TCPV4_WAN** (9)

  IPv4 TCP/IP Transport-Plugin class ID for WAN case.

- #define **NDDS_TRANSPORT_CLASSNAME_TCPV4_WAN** "tcpv4_wan"

  IPv4 TCP/IP Transport-Plugin class name for WAN case.

- #define **NDDS_TRANSPORT_CLASSID_TLSV4_LAN** (10)

  IPv4 TCP/IP Transport-Plugin class ID for LAN case with TLS enabled.

- #define **NDDS_TRANSPORT_CLASSID_TLSV4_WAN** (11)

  IPv4 TCP/IP Transport-Plugin class ID for WAN case with TLS enabled.

- #define **NDDS_TRANSPORT_CLASSID_PCIE** (12)

  PCIE Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_CLASSID_ITP** (13)

  Internet Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_CLASSID_RESERVED_RANGE** (1000)

  Transport-Plugin class IDs below this are reserved by RTI.

Typedefs

- typedef RTI_INT32 **NDDS_Transport_ClassId_t**

  Type for storing RTI Connext Transport-Plugin class IDs.

5.62.1 Detailed Description

Transport class ID. Specifies the ID of a Transport-Plugin 'class'.

Each implementation of a Transport Plugin must have a unique ID. For example, a UDP/IP Transport-Plugin implemenation must have a different ID than a Shared Memory Transport Plugin.

User-implemented Transport Plugins must have an ID higher than **NDDS_TRANSPORT_CLASSID_RESERVED_RANGE** (p. 249).

5.62.2 Define Documentation

5.62.2.1 #define NDDS_TRANSPORT_CLASSID_INVALID (-1)

Invalid Transport Class ID.

Transport-Plugins implementations should set their class ID to a value different than this.

5.62.2.2 #define NDDS_TRANSPORT_CLASSID_DTLS (6)

DTLS Secure Transport-Plugin class ID.

5.62.2.3 #define NDDS_TRANSPORT_CLASSID_WAN (7)

WAN Transport-Plugin class ID.

5.62.2.4 #define NDDS_TRANSPORT_CLASSID_TCPV4_LAN (8)

IPv4 TCP/IP Transport-Plugin class ID for LAN case.

5.62.2.5 #define NDDS_TRANSPORT_CLASSID_TCPV4_WAN (9)

IPv4 TCP/IP Transport-Plugin class ID for WAN case.

5.62.2.6 #define NDDS_TRANSPORT_CLASSNAME_TCPV4_WAN "tcpv4_wan"

IPv4 TCP/IP Transport-Plugin class name for WAN case.

5.62.2.7 #define NDDS_TRANSPORT_CLASSID_TLSV4_LAN (10)

IPv4 TCP/IP Transport-Plugin class ID for LAN case with TLS enabled.

5.62.2.8 #define NDDS_TRANSPORT_CLASSID_TLSV4_WAN (11)

IPv4 TCP/IP Transport-Plugin class ID for WAN case with TLS enabled.

5.62.2.9 #define NDDS_TRANSPORT_CLASSID_PCIE (12)

PCIE Transport-Plugin class ID.

5.62.2.10 #define NDDS_TRANSPORT_CLASSID_ITP (13)

Internet Transport-Plugin class ID.

5.62.2.11 #define NDDS_TRANSPORT_CLASSID_RESERVED_RANGE (1000)

Transport-Plugin class IDs below this are reserved by RTI.

User-defined Transport-Plugins should use a class ID greater than this number.

5.62.3 Typedef Documentation

5.62.3.1 typedef RTI_INT32 NDDS_Transport_ClassId_t

Type for storing RTI Connext Transport-Plugin class IDs.

5.63 Address

Transport-independent addressing scheme using IPv6 presentation strings and numerically stored in network-ordered format.

Classes

- struct **NDDS_Transport_Address_t**

  Addresses are stored individually as network-ordered bytes.

Defines

- #define **NDDS_TRANSPORT_ADDRESS_INVALID** {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}

  An invalid transport address. Used as an initializer.

- #define **NDDS_TRANSPORT_ADDRESS_STRING_BUFFER_SIZE** (40)

  *The minimum size of the buffer that should be passed to **NDDS_Transport_Address_to_string** (p. 252).*

Functions

- RTI_INT32 **NDDS_Transport_Address_to_string** (const **NDDS_Transport_Address_t** *self, char *buffer_inout, RTI_INT32 buffer_length_in)

  Converts a numerical address to a printable string representation.

- RTI_INT32 **NDDS_Transport_Address_from_string** (**NDDS_Transport_Address_t** *address_out, const char *address_in)

  Converts an address (IPv4 dotted notation or IPv6 presentation string) into a numerical address.

- void **NDDS_Transport_Address_print** (const **NDDS_Transport_Address_t** *address_in, const char *desc_in, RTI_INT32 indent_in)

  Prints an address to standard out.

- RTI_INT32 **NDDS_Transport_Address_is_ipv4** (const **NDDS_Transport_Address_t** *address_in)

  Checks if an address is an IPv4 address.

- RTI_INT32 **NDDS_Transport_Address_is_multicast** (const **NDDS_Transport_Address_t** *address_in)

  Checks if an address is an IPv4 or IPv6 multicast address.

5.63.1 Detailed Description

Transport-independent addressing scheme using IPv6 presentation strings and numerically stored in network-ordered format.

The APIs of RTI Connext uses IPv6 address notation for all transports.

Transport-Plugin implementations that are not IP-based are required to map whatever addressing scheme natively used by the physical transport (if any) to an address in IPv6 notation and vice versa.

IPv6 addresses are numerically stored in 16 bytes. An IPv6 address can be presented In string notation in a variety of ways. For example,

```
"00AF:0000:0037:FE01:0000:0000:034B:0089"
"AF:0:37:FE01:0:0:34B:89"
"AF:0:37:FE01::34B:89"
```

are all valid IPv6 presentation of the same address.

IPv4 address in dot notation can be used to specify the last 4 bytes of the address. For example,

```
"0000:0000:0000:0000:0000:0000:192.168.0.1"
"0:0:0:0:0:0:192.168.0.1"
"::192.168.0.1"
```

are all valid IPv6 presentation of the same address.

For a complete description of valid IPv6 address notation, consult the IPv6 Addressing Architecture (RFC 2373).

Addresses are divided into unicast addresses and multicast addresses.

Multicast addresses are defined as

- Addresses that start with 0xFF. That is `FFxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx`.

or an IPv4 multicast address

- Address in the range [::224.0.0.0, ::239.255.255.255]

Multicast addresses do not refer to any specific destination (network interface). Instead, they usually refer to a group of network interfaces, often called a "multicast group".

Unicast addresses always refer to a specific network interface.

5.63.2 Define Documentation

5.63.2.1 #define NDDS_TRANSPORT_ADDRESS_INVALID {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}

An invalid transport address. Used as an initializer.

For example: **NDDS_Transport_Address_t** (p. 1521) address = NDDS_TRANSPORT_ADDRESS_INVALID

5.63.2.2 #define NDDS_TRANSPORT_ADDRESS_STRING_BUFFER_SIZE (40)

The minimum size of the buffer that should be passed to **NDDS_Transport_Address_to_string** (p. 252).

The string size includes space for 8 tuples of 4 characters each plus 7 delimiting colons plus a terminating NULL.

5.63.3 Function Documentation

5.63.3.1 RTI_INT32 NDDS_Transport_Address_to_string (const NDDS_Transport_Address_t * *self*, char * *buffer_inout*, RTI_INT32 *buffer_length_in*)

Converts a numerical address to a printable string representation.

Precondition:

The `buffer_inout` provided must be at least **NDDS_TRANSPORT_ADDRESS_STRING_BUFFER_SIZE** (p. 252) characters long.

Parameters:

self *<<in>> (p. 200)* The address to be converted.

buffer_inout *<<inout>> (p. 200)* Storage passed in which to return the string corresponding to the address.

buffer_length_in *<<in>> (p. 200)* The length of the storage buffer. Must be >= **NDDS_TRANSPORT_ADDRESS_STRING_BUFFER_SIZE** (p. 252)

Returns:

1 upon success; 0 upon failure (not enough space in the provided buffer)

5.63.3.2 RTI_INT32 NDDS_Transport_Address_from_string (NDDS_Transport_Address_t * *address_out*, const char * *address_in*)

Converts an address (IPv4 dotted notation or IPv6 presentation string) into a numerical address.

The address string must be in IPv4 dotted notation (X.X.X.X) or IPv6 presentation notation. The string cannot be a hostname since this function does not perform a hostname lookup.

Parameters:

address_out *<<**out**>>* *(p. 200)* Numerical value of the address.

address_in *<<**in**>>* *(p. 200)* String representation of an address.

Returns:

1 if address_out contains a valid address
0 if it was not able to convert the string into an address.

5.63.3.3 void NDDS_Transport_Address_print (const NDDS_Transport_Address_t * *address_in*, const char * *desc_in*, RTI_INT32 *indent_in*)

Prints an address to standard out.

Parameters:

address_in *<<**in**>>* *(p. 200)* Address to be printed.

desc_in *<<**in**>>* *(p. 200)* A prefix to be printed before the address.

indent_in *<<**in**>>* *(p. 200)* Indentation level for the printout.

5.63.3.4 RTI_INT32 NDDS_Transport_Address_is_ipv4 (const NDDS_Transport_Address_t * *address_in*)

Checks if an address is an IPv4 address.

Parameters:

address_in *<<**in**>>* *(p. 200)* Address to be tested.

Note:

May be implemented as a macro for efficiency.

Returns:

1 if address is an IPv4 address
0 otherwise.

5.63.3.5 RTI_INT32 NDDS_Transport_Address_is_multicast (const NDDS_Transport_Address_t * *address_in*)

Checks if an address is an IPv4 or IPv6 multicast address.

Parameters:

address_in <<*in*>> *(*p. 200*)* Address to be tested.

May be implemented as a macro for efficiency.

Returns:

1 if address is a multicast address
0 otherwise.

5.64 Attributes

Base "class" of the properties of any Transport Plugin.

Classes

- struct **NDDS_Transport_Property_t**

  Base structure that must be inherited by derived Transport Plugin classes.

Defines

- #define **NDDS_TRANSPORT_PROPERTY_BIT_BUFFER_ALWAYS_LOANED** (0x2)

  Specified zero-copy behavior of transport.

- #define **NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN** (3)

  Minimum number of gather-send buffers that must be supported by a Transport Plugin implementation.

5.64.1 Detailed Description

Base "class" of the properties of any Transport Plugin.

5.64.2 Define Documentation

5.64.2.1 #define NDDS_TRANSPORT_PROPERTY_BIT_BUFFER_ALWAYS_LOANED (0x2)

Specified zero-copy behavior of transport.

A Transport Plugin may commit to one of three behaviors for zero copy receives:

1. Always does zero copy.
2. Sometimes does zero copy, up to the transport discretion.
3. Never does zero copy.

This bit should be set only if the Transport Plugin commits to always doing a zero copy receive, or more specifically, always loaning a buffer through its `receive_rEA()` call.

In that case, the NDDS core will not need to allocate storage for a message that it retrieves with the `receive_rEA()` call.

5.64.2.2 #define NDDS_TRANSPORT_PROPERTY_GATHER_-SEND_BUFFER_COUNT_MIN (3)

Minimum number of gather-send buffers that must be supported by a Transport Plugin implementation.

For the **NDDS_Transport_Property_t** (p. 1522) structure to be valid, the value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525) must be greater than or equal to this value.

5.65 Shared Memory Transport

Built-in transport plug-in for inter-process communications using shared memory (**NDDS_TRANSPORT_CLASSID_SHMEM** (p. 261)).

Classes

- struct **NDDS_Transport_Shmem_Property_t**

  *Subclass of **NDDS_Transport_Property_t** (p. 1522) allowing specification of parameters that are specific to the shared-memory transport.*

Defines

- #define **NDDS_TRANSPORT_CLASSID_SHMEM** (2)

  Builtin Shared-Memory Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_SHMEM_ADDRESS_BIT_COUNT** (0)

  *Default value of **NDDS_Transport_Property_t::address_bit_count** (p. 1524).*

- #define **NDDS_TRANSPORT_SHMEM_PROPERTIES_BITMAP_DEFAULT** (NDDS_TRANSPORT_PROPERTY_BIT_BUFFER_ALWAYS_LOANED)

  *Default value of **NDDS_Transport_Property_t::properties_bitmap** (p. 1524).*

- #define **NDDS_TRANSPORT_SHMEM_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT** (1024)

  *Default value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).*

- #define **NDDS_TRANSPORT_SHMEM_MESSAGE_SIZE_MAX_DEFAULT** (9216)

  *Default value of **NDDS_Transport_Property_t::message_size_max** (p. 1525).*

- #define **NDDS_TRANSPORT_SHMEM_RECEIVED_MESSAGE_COUNT_MAX_DEFAULT** (32)

  *Default value of **NDDS_Transport_Shmem_Property_t::received_message_count_max** (p. 1530).*

- #define **NDDS_TRANSPORT_SHMEM_RECEIVE_BUFFER_SIZE_DEFAULT**

  *Default value of **NDDS_Transport_Shmem_Property_t::receive_buffer_size** (p. 1531).*

- #define **NDDS_TRANSPORT_SHMEM_PROPERTY_DEFAULT**

  Use this to initialize stack variable.

Functions

- NDDS_Transport_Plugin * **NDDS_Transport_Shmem_new** (const struct **NDDS_Transport_Shmem_Property_t** *property_in)

  Create a new shmem process transport.

- NDDS_Transport_Plugin * **NDDS_Transport_Shmem_create** (**NDDS_Transport_Address_t** *default_network_address_out, const struct **DDS_PropertyQosPolicy** *property_in)

  Create a new shmem process transport, using PropertyQosPolicy.

5.65.1 Detailed Description

Built-in transport plug-in for inter-process communications using shared memory (**NDDS_TRANSPORT_CLASSID_SHMEM** (p. 261)).

This plugin uses System Shared Memory to send messages between processes on the same node.

The transport plugin has exactly one "receive interface"; since the `address_bit_count` is 0, it can be assigned any address. Thus the interface is located by the "network address" associated with the transport plugin.

5.65.2 Compatibility of Sender and Receiver Transports

Opening a receiver "port" on shared memory corresponds to creating a shared memory segment using a name based on the port number. The transport plugin's properties are embedded in the shared memory segment.

When a sender tries to send to the shared memory port, it verifies that properties of the receiver's shared memory transport are compatible with those specified in its transport plugin. If not, the sender will fail to attach to the port and will output messages such as below (with numbers appropriate to the properties of the transport plugins involved).

```
NDDS_Transport_Shmem_attachShmem:failed to initialize incompatible properties
NDDS_Transport_Shmem_attachShmem:countMax 0 > -19417345 or max size -19416188 > 2147482624
```

In this scenario, the properties of the sender or receiver transport plugin instances should be adjusted, so that they are compatible.

5.65.3 Crashing and Restarting Programs

If a process using shared memory crashes (say because the user typed in ^C), resources associated with its shared memory ports may not be properly cleaned up. Later, if another RTI Connext process needs to open the same ports (say, the crashed program is restarted), it will attempt to reuse the shared memory segment left behind by the crashed process.

The reuse is allowed iff the properties of transport plugin are compatible with those embedded in the shared memory segment (i.e., of the original creator). Otherwise, the process will fail to open the ports, and will output messages such as below (with numbers appropriate to the properties of the transport plugins involved).

```
NDDS_Transport_Shmem_create_recvresource_rrEA:failed to initialize shared
 memory resource Cannot recycle existing shmem: size not compatible for key 0x1234
```

In this scenario, the shared memory segments must be cleaned up using appropriate platform specific commands. For details, please refer to the `Platform Notes`.

5.65.4 Shared Resource Keys

The transport uses the **shared memory segment keys**, given by the formula below.

```
0x400000 + port
```

The transport also uses signaling **shared semaphore keys** given by the formula below.

```
0x800000 + port
```

The transport also uses mutex **shared semaphore keys** given by the formula below.

```
0xb00000 + port
```

wher the `port` is a function of the `domain_id` and the `participant_id`, as described in **DDS_WireProtocolQosPolicy::participant_id** (p. 1063)

See also:

DDS_WireProtocolQosPolicy::participant_id (p. 1063)
NDDSTransportSupport::set_builtin_transport_property() (p. 1564)

5.65.5 Creating and Registering Shared Memory Transport Plugin

RTI Connext can implicitly create this plugin and register with the **DDSDomainParticipant** (p. 1139) if this transport is specified in **DDS_TransportBuiltinQosPolicy** (p. 969).

To specify the properties of the builtin shared memory transport that is implicitly registered, you can either:

- call **NDDSTransportSupport::set_builtin_transport_property** (p. 1564) or
- specify the pre-defined property names in **DDS_PropertyQosPolicy** (p. 834) associated with the **DDSDomainParticipant** (p. 1139). (see **Shared Memory Transport Property Names in Property QoS Policy of Domain Participant** (p. 261)).

Builtin transport plugin properties specified in **DDS_PropertyQosPolicy** (p. 834) always overwrite the ones specified through **NDDSTransportSupport::set_builtin_transport_property()** (p. 1564). The default value is assumed on any unspecified property. Note that all properties should be set before the transport is implicitly created and registered by RTI Connext. See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

To explicitly create an instance of this plugin, **NDDS_Transport_Shmem_new()** (p. 262) should be called. The instance should be registered with RTI Connext, see **NDDSTransportSupport::register_transport** (p. 1560). In some configurations, you may have to disable the builtin shared memory transport plugin instance (**DDS_TransportBuiltinQosPolicy** (p. 969), **DDS_TRANSPORTBUILTIN_SHMEM** (p. 398)), to avoid port conflicts with the newly created plugin instance.

5.65.6 Shared Memory Transport Property Names in Property QoS Policy of Domain Participant

The following table lists the predefined property names that can be set in the **DDS_PropertyQosPolicy** (p. 834) of a **DDSDomainParticipant** (p. 1139) to configure the builtin shared memory transport plugin.

5.65.7 Define Documentation

5.65.7.1 #define NDDS_TRANSPORT_CLASSID_SHMEM (2)

Builtin Shared-Memory Transport-Plugin class ID.

5.65.7.2 #define NDDS_TRANSPORT_SHMEM_ADDRESS_BIT_COUNT (0)

Default value of **NDDS_Transport_Property_t::address_bit_count** (p. 1524).

5.65.7.3 #define NDDS_TRANSPORT_SHMEM_PROPERTIES_BITMAP_DEFAULT (NDDS_TRANSPORT_PROPERTY_BIT_BUFFER_ALWAYS_LOANED)

Default value of **NDDS_Transport_Property_t::properties_bitmap** (p. 1524).

5.65.7.4 #define NDDS_TRANSPORT_SHMEM_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT (1024)

Default value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).

5.65.7.5 #define NDDS_TRANSPORT_SHMEM_MESSAGE_SIZE_MAX_DEFAULT (9216)

Default value of **NDDS_Transport_Property_t::message_size_max** (p. 1525).

5.65.7.6 #define NDDS_TRANSPORT_SHMEM_RECEIVED_MESSAGE_COUNT_MAX_DEFAULT (32)

Default value of **NDDS_Transport_Shmem_Property_t::received_message_count_max** (p. 1530).

5.65.7.7 #define NDDS_TRANSPORT_SHMEM_RECEIVE_BUFFER_SIZE_DEFAULT

Value:

```
(NDDS_TRANSPORT_SHMEM_RECEIVED_MESSAGE_COUNT_MAX_DEFAULT * \
        NDDS_TRANSPORT_SHMEM_MESSAGE_SIZE_MAX_DEFAULT / 4)
```

Default value of **NDDS_Transport_Shmem_Property_t::receive_buffer_size** (p. 1531).

5.65.7.8 #define NDDS_TRANSPORT_SHMEM_PROPERTY_DEFAULT

Use this to initialize stack variable.

5.65.8 Function Documentation

5.65.8.1 NDDS_Transport_Plugin* NDDS_Transport_Shmem_new (const struct NDDS_Transport_Shmem_Property_t * *property_in*)

Create a new shmem process transport.

An application may create multiple transports, possibly for use in different domains.

Parameters:

property_in *<<in>> (p. 200)* Desired behavior of this transport. May be NULL for default property. The transport plugin can only support one unicast receive interface; therefore the interface selection lists are ignored.

Returns:

handle to a Shmem inter-process Transport Plugin on success
NULL on failure.

5.65.8.2 NDDS_Transport_Plugin* NDDS_Transport_Shmem_create (NDDS_Transport_Address_t * *default_network_address_out*, const struct DDS_PropertyQosPolicy * *property_in*)

Create a new shmem process transport, using PropertyQosPolicy.

An application may create multiple transports, possibly for use in different domains.

Parameters:

default_network_address_out <<*out*>> *(*p. 200*)* Network address to be used when registering the transport.

property_in <<*in*>> *(*p. 200*)* Desired behavior of this transport. May be NULL for default property. The transport plugin can only support one unicast receive interface; therefore the interface selection lists are ignored.

Returns:

handle to a Shmem inter-process Transport Plugin on success
NULL on failure.

Name	Descriptions
dds.transport.shmem.builtin.parent.address_bit_count	See **NDDS_Transport_Property_t::address_bit_count** (p. 1524)
dds.transport.shmem.builtin.parent.properties_bitmap	See **NDDS_Transport_Property_t::properties_bitmap** (p. 1524)
dds.transport.shmem.builtin.parent.gather_send_buffer_count_max	See **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525)
dds.transport.shmem.builtin.parent.message_size_max	See **NDDS_Transport_Property_t::message_size_max** (p. 1525)
dds.transport.shmem.builtin.parent.allow_interfaces	See **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526) and **NDDS_Transport_Property_t::allow_interfaces_list_length** (p. 1526). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.shmem.builtin.parent.deny_interfaces	See **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526) and **NDDS_Transport_Property_t::deny_interfaces_list_length** (p. 1527). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.shmem.builtin.parent.allow_multicast_interfaces	See **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527) and **NDDS_Transport_Property_t::allow_multicast_interfaces_list_length** (p. 1528). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.shmem.builtin.parent.deny_multicast_interfaces	See **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528) and **NDDS_Transport_Property_t::deny_multicast_interfaces_list_length** (p. 1528). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0

5.66 UDPv4 Transport

Built-in transport plug-in using UDP/IPv4 (**NDDS_TRANSPORT_CLASSID_UDPv4** (p. 268)).

Classes

- struct **NDDS_Transport_UDPv4_Property_t**

  Configurable IPv4/UDP Transport-Plugin properties.

Defines

- #define **NDDS_TRANSPORT_CLASSID_UDPv4** (1)

  Builtin IPv4 UDP/IP Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_UDPV4_ADDRESS_BIT_COUNT** (32)

  *Default value of **NDDS_Transport_Property_t::address_bit_count** (p. 1524).*

- #define **NDDS_TRANSPORT_UDPV4_PROPERTIES_BITMAP_DEFAULT** (0)

  *Default value of **NDDS_Transport_Property_t::properties_bitmap** (p. 1524).*

- #define **NDDS_TRANSPORT_UDPV4_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT** (16)

  *Default value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).*

- #define **NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT** (-1)

  Used to specify that os default be used to specify socket buffer size.

- #define **NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT** (9216)

  *Default value of **NDDS_Transport_Property_t::message_size_max** (p. 1525).*

- #define **NDDS_TRANSPORT_UDPV4_MULTICAST_TTL_DEFAULT** (1)

*Default value of **NDDS_Transport_UDPv4_Property_t::multicast_ttl*** (p. 1536).

- #define **NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER**

  *Value for **NDDS_Transport_UDPv4_Property_t::send_blocking*** (p. 1539) *to specify non-blocking sockets.*

- #define **NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS**

  [default] *Value for **NDDS_Transport_UDPv4_Property_t::send_blocking*** (p. 1539) *to specify blocking sockets.*

- #define **NDDS_TRANSPORT_UDPV4_BLOCKING_DEFAULT** NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS

  *Default value for **NDDS_Transport_UDPv4_Property_t::send_blocking*** (p. 1539) *to specify blocking sockets.*

- #define **NDDS_TRANSPORT_UDPV4_PROPERTY_DEFAULT**

  *Use this to initialize a **NDDS_Transport_UDPv4_Property_t*** (p. 1533) *structure.*

Functions

- NDDS_Transport_Plugin * **NDDS_Transport_UDPv4_new** (const struct **NDDS_Transport_UDPv4_Property_t** *property_in)

  Create an instance of a UDPv4 Transport Plugin.

- RTI_INT32 **NDDS_Transport_UDPv4_string_to_address_cEA** (NDDS_Transport_Plugin *self, **NDDS_Transport_Address_t** *address_out, const char *address_in)

  Realization of NDDS_Transport_String_To_Address_Fcn_cEA for IP transports.

- RTI_INT32 **NDDS_Transport_UDPv4_get_receive_interfaces_cEA** (NDDS_Transport_Plugin *self, RTI_INT32 *found_more_than_provided_for_out, RTI_INT32 *interface_reported_count_out, NDDS_Transport_Interface_t interface_array_inout[], RTI_INT32 interface_array_size_in)

  Realization of NDDS_Transport_Get_Receive_Interfaces_Fcn_cEA for IP transports.

- NDDS_Transport_Plugin * **NDDS_Transport_UDPv4_create** (**NDDS_Transport_Address_t** *default_network_address_out, const struct **DDS_PropertyQosPolicy** *property_in)

Create an instance of a UDPv4 Transport Plugin, using PropertyQosPolicy.

5.66.1 Detailed Description

Built-in transport plug-in using UDP/IPv4 (**NDDS_TRANSPORT_CLASSID_UDPv4** (p. 268)).

This transport plugin uses UDPv4 sockets to send and receive messages. It supports both unicast and multicast communications in a single instance of the plugin. By default, this plugin will use all interfaces that it finds enabled and "UP" at instantiation time to send and receive messages.

The user can configure an instance of this plugin to only use unicast or only use multicast, see **NDDS_Transport_UDPv4_Property_t::unicast_enabled** (p. 1536) and **NDDS_Transport_UDPv4_Property_t::multicast_enabled** (p. 1536).

In addition, the user can configure an instance of this plugin to selectively use the network interfaces of a node (and restrict a plugin from sending multicast messages on specific interfaces) by specifying the "white" and "black" lists in the base property's fields (**NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526), **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526), **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527), **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528)).

RTI Connext can implicitly create this plugin and register with the **DDSDomainParticipant** (p. 1139) if this transport is specified in **DDS_TransportBuiltinQosPolicy** (p. 969).

To specify the properties of the builtin UDPv4 transport that is implicitly registered, you can either:

- call **NDDSTransportSupport::set_builtin_transport_property** (p. 1564) or
- specify the predefined property names in **DDS_PropertyQosPolicy** (p. 834) associated with the **DDSDomainParticipant** (p. 1139). (see **UDPv4 Transport Property Names in Property QoS Policy of Domain Participant** (p. 268)). Builtin transport plugin properties specified in **DDS_PropertyQosPolicy** (p. 834) always overwrite the ones specified through **NDDSTransportSupport::set_builtin_transport_property()** (p. 1564). The default value is assumed on any unspecified property.

Note that all properties should be set before the transport is implicitly created and registered by RTI Connext. Any properties set after the builtin

transport is registered will be ignored. See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered. To explicitly create an instance of this plugin, **NDDS_Transport_UDPv4_new()** (p. 270) should be called. The instance should be registered with RTI Connext, see **NDDSTransportSupport::register_transport** (p. 1560). In some configurations one may have to disable the builtin UDPv4 transport plugin instance (**DDS_TransportBuiltinQosPolicy** (p. 969), **DDS_TRANSPORTBUILTIN_UDPv4** (p. 398)), to avoid port conflicts with the newly created plugin instance.

5.66.2 UDPv4 Transport Property Names in Property QoS Policy of Domain Participant

The following table lists the predefined property names that can be set in **DDS_PropertyQosPolicy** (p. 834) of a **DDSDomainParticipant** (p. 1139) to configure the builtin UDPv4 transport plugin.

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564)

5.66.3 Define Documentation

5.66.3.1 #define NDDS_TRANSPORT_CLASSID_UDPv4 (1)

Builtin IPv4 UDP/IP Transport-Plugin class ID.

5.66.3.2 #define NDDS_TRANSPORT_UDPV4_ADDRESS_BIT_COUNT (32)

Default value of **NDDS_Transport_Property_t::address_bit_count** (p. 1524).

5.66.3.3 #define NDDS_TRANSPORT_UDPV4_PROPERTIES_BITMAP_DEFAULT (0)

Default value of **NDDS_Transport_Property_t::properties_bitmap** (p. 1524).

5.66.3.4 #define NDDS_TRANSPORT_UDPV4_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT (16)

Default value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).

This is also the maximum value that can be used when instantiating the udp transport.

16 is sufficient for RTI Connext, but more may improve discovery and reliable performance. Porting note: find out what the maximum gather buffer count is on your OS!

5.66.3.5 #define NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT (-1)

Used to specify that os default be used to specify socket buffer size.

5.66.3.6 #define NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT (9216)

Default value of **NDDS_Transport_Property_t::message_size_max** (p. 1525).

5.66.3.7 #define NDDS_TRANSPORT_UDPV4_MULTICAST_TTL_DEFAULT (1)

Default value of **NDDS_Transport_UDPv4_Property_t::multicast_ttl** (p. 1536).

5.66.3.8 #define NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER

Value for **NDDS_Transport_UDPv4_Property_t::send_blocking** (p. 1539) to specify non-blocking sockets.

5.66.3.9 #define NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS

[**default**] Value for **NDDS_Transport_UDPv4_Property_t::send_blocking** (p. 1539) to specify blocking sockets.

5.66.3.10 #define NDDS_TRANSPORT_UDPV4_BLOCKING_DEFAULT NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS

Default value for **NDDS_Transport_UDPv4_Property_t::send_blocking** (p. 1539) to specify blocking sockets.

5.66.3.11 #define NDDS_TRANSPORT_UDPV4_PROPERTY_DEFAULT

Value:

```
{ \
    { NDDS_TRANSPORT_CLASSID_UDPv4, \
      NDDS_TRANSPORT_UDPV4_ADDRESS_BIT_COUNT, \
      NDDS_TRANSPORT_UDPV4_PROPERTIES_BITMAP_DEFAULT, \
      NDDS_TRANSPORT_UDPV4_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT, \
      NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT, \
      NULL, 0, /* allow_interfaces_list */ \
      NULL, 0, /* deny_interfaces_list */ \
      NULL, 0, /* allow_multicast_interfaces_list */ \
      NULL, 0, /* deny_multicast_interfaces_list */ \
    }, \
    NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT, \
    NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT, \
    1, /* use unicast */   \
    NDDS_TRANSPORT_UDPV4_USE_MULTICAST_DEFAULT, /* use multicast */ \
    NDDS_TRANSPORT_UDPV4_MULTICAST_TTL_DEFAULT, \
    0, /* multicast loopback enabled */ \
    -1, /* (auto-)ignore loopback */ \
    1, /* ignore_nonup_interfaces */ \
    0, /* do not ignore non-RUNNING */ \
    0, /* no_zero_copy */ \
    NDDS_TRANSPORT_UDPV4_BLOCKING_DEFAULT, \
    0, 0, 0xff /* no mapping to IP_TOS by default */, \
    1, /* send_ping */ \
    500, /* 500 millisecs is the default polling period*/ \
    1, /* reuse multicast receive resource */ \
    28 /* protocol_overhead_max (UDP(8)+IP(20)) */}
```

Use this to initialize a **NDDS_Transport_UDPv4_Property_t** (p. 1533) structure.

5.66.4 Function Documentation

5.66.4.1 NDDS_Transport_Plugin* NDDS_Transport_UDPv4_new (const struct NDDS_Transport_UDPv4_Property_t * *property_in*)

Create an instance of a UDPv4 Transport Plugin.

An application may create and register multiple instances of this Transport Plugin with RTI Connext. This may be to partition the network interfaces across multiple RTI Connext domains. However, note that the underlying transport, the operating system's IP layer, is still a "singleton". For example, if a unicast transport has already bound to a port, and another unicast transport tries to bind to the same port, the second attempt will fail.

The transport plugin honors the interface/multicast "white" and "black" lists specified in the **NDDS_Transport_UDPv4_Property_t::parent** (p. 1534):

- **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527),
- **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528)

The format of a string in these lists is assumed to be in standard IPv4 dot notation, possibly containing wildcards. For example:

- 10.10.30.232
- 10.10.*.*
- 192.168.1.*
- etc. Strings not in the correct format will be ignored.

Parameters:

property_in *<<in>>* *(p. 200)* Desired behavior of this transport. May be NULL for default property.

Returns:

A UDPv4 Transport Plugin instance on success; or NULL on failure.

5.66.4.2 RTI_INT32 NDDS_Transport_UDPv4_string_to_address_cEA (NDDS_Transport_Plugin * *self*, NDDS_Transport_Address_t * *address_out*, const char * *address_in*)

Realization of NDDS_Transport_String_To_Address_Fcn_cEA for IP transports.

Converts a host name string to a IPv4 address.

Parameters:

self NOT USED. May be NULL.

address_out <<***out***>> *(*p. *200)* The corresponding numerical value in IPv6 format.

address_in <<***in***>> *(*p. *200)* The name of the IPv4 address. It can be a dot notation name or a host name. If NULL, then the IP address of the localhost will be returned.

See also:

NDDS_Transport_String_To_Address_Fcn_cEA for complete documentation.

5.66.4.3 RTI_INT32 NDDS_Transport_UDPv4_get_receive_interfaces_cEA (NDDS_Transport_Plugin * *self*, RTI_INT32 * *found_more_than_provided_for_out*, RTI_INT32 * *interface_reported_count_out*, NDDS_Transport_Interface_t *interface_array_inout*[], RTI_INT32 *interface_array_size_in*)

Realization of NDDS_Transport_Get_Receive_Interfaces_Fcn_cEA for IP transports.

Retrieves a list of available IPv4 network interfaces. The addresses returned from IPv4 plugin will be of the pattern 0.0.0.0.0.0.0.0.0.0.0.0.x.x.x.x.

See also:

NDDS_Transport_Get_Receive_Interfaces_Fcn_cEA for complete documentation.

5.66.4.4 NDDS_Transport_Plugin* NDDS_Transport_UDPv4_create (NDDS_Transport_Address_t * *default_network_address_out*, const struct DDS_PropertyQosPolicy * *property_in*)

Create an instance of a UDPv4 Transport Plugin, using PropertyQosPolicy.

An application may create and register multiple instances of this Transport Plugin with RTI Connext. This may be to partition the network interfaces across multiple RTI Connext domains. However, note that the underlying transport, the operating system's IP layer, is still a "singleton". For example, if a unicast transport has already bound to a port, and another unicast transport tries to bind to the same port, the second attempt will fail.

The transport plugin honors the interface/multicast "white" and "black" lists specified in the **NDDS_Transport_UDPv4_Property_t::parent** (p. 1534):

- **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527),
- **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528)

The format of a string in these lists is assumed to be in standard IPv4 dot notation, possibly containing wildcards. For example:

- 10.10.30.232
- 10.10.*.*
- 192.168.1.*
- etc. Strings not in the correct format will be ignored.

Parameters:

default_network_address_out *<<out>> (p. 200)* Network address to be used when registering the transport.

property_in *<<in>> (p. 200)* Desired behavior of this transport, through the property field in **DDS_DomainParticipantQos** (p. 588).

Returns:

A UDPv4 Transport Plugin instance on success; or NULL on failure.

Property Name	Description
dds.transport.UDPv4.builtin.parent.address_bit_count	See **NDDS_Transport_Property_t::address_bit_count** (p. 1524)
dds.transport.UDPv4.builtin.parent.properties_bitmap	See **NDDS_Transport_Property_t::properties_bitmap** (p. 1524)
dds.transport.UDPv4.builtin.parent.gather_send_buffer_count_max	See **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525)
dds.transport.UDPv4.builtin.parent.message_size_max	See **NDDS_Transport_Property_t::message_size_max** (p. 1525)
dds.transport.UDPv4.builtin.parent.allow_interfaces	See **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526) and **NDDS_Transport_Property_t::allow_interfaces_list_length** (p. 1526). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, 127.0.0.1,eth0
dds.transport.UDPv4.builtin.parent.deny_interfaces	See **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526) and **NDDS_Transport_Property_t::deny_interfaces_list_length** (p. 1527). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.UDPv4.builtin.parent.allow_multicast_interfaces	See **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527) and **NDDS_Transport_Property_t::allow_multicast_interfaces_list_length** (p. 1528). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.UDPv4.builtin.parent.deny_multicast_interfaces	See **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528) and **NDDS_Transport_Property_t::deny_multicast_interfaces_list_length** (p. 1528). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0

5.67 UDPv6 Transport

Built-in transport plug-in using UDP/IPv6 (**NDDS_TRANSPORT_CLASSID_UDPv6** (p. 278)).

Classes

- struct **NDDS_Transport_UDPv6_Property_t**

  Configurable IPv6/UDP Transport-Plugin properties.

Defines

- #define **NDDS_TRANSPORT_CLASSID_UDPv6** (5)

  Builtin IPv6 UDP/IP Transport-Plugin class ID.

- #define **NDDS_TRANSPORT_UDPV6_ADDRESS_BIT_COUNT** (128)

  *Default value of **NDDS_Transport_Property_t::address_bit_count** (p. 1524).*

- #define **NDDS_TRANSPORT_UDPV6_PROPERTIES_BITMAP_DEFAULT** (0)

  *Default value of **NDDS_Transport_Property_t::properties_bitmap** (p. 1524).*

- #define **NDDS_TRANSPORT_UDPV6_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT** (16)

  *Default value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).*

- #define **NDDS_TRANSPORT_UDPV6_SOCKET_BUFFER_SIZE_OS_DEFAULT** (-1)

  Used to specify that os default be used to specify socket buffer size.

- #define **NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_DEFAULT** (9216)

  *Default value of **NDDS_Transport_Property_t::message_size_max** (p. 1525).*

- #define **NDDS_TRANSPORT_UDPV6_MULTICAST_TTL_DEFAULT** (1)

*Default value of **NDDS_Transport_UDPv6_Property_t::multicast_ttl*** *(p. 1545).*

- #define **NDDS_TRANSPORT_UDPV6_BLOCKING_NEVER**

  *Value for **NDDS_Transport_UDPv6_Property_t::send_blocking*** *(p. 1547) to specify non-blocking sockets.*

- #define **NDDS_TRANSPORT_UDPV6_BLOCKING_ALWAYS**

  [default] *Value for **NDDS_Transport_UDPv6_Property_t::send_blocking*** *(p. 1547) to specify blocking sockets.*

- #define **NDDS_TRANSPORT_UDPV6_PROPERTY_DEFAULT**

  *Use this to initialize a **NDDS_Transport_UDPv6_Property_t*** *(p. 1542) structure.*

Functions

- NDDS_Transport_Plugin * **NDDS_Transport_UDPv6_new** (const struct **NDDS_Transport_UDPv6_Property_t** *property_in)

  Create an instance of a UDPv6 Transport Plugin.

- RTI_INT32 **NDDS_Transport_UDPv6_string_to_address_cEA** (NDDS_Transport_Plugin *self, **NDDS_Transport_Address_t** *address_out, const char *address_in)

  Realization of NDDS_Transport_String_To_Address_Fcn_cEA for IP transports.

- RTI_INT32 **NDDS_Transport_UDPv6_get_receive_interfaces_cEA** (NDDS_Transport_Plugin *self, RTI_INT32 *found_more_than_provided_for_out, RTI_INT32 *interface_reported_count_out, NDDS_Transport_Interface_t interface_array_inout[], RTI_INT32 interface_array_size_in)

  Realization of NDDS_Transport_Get_Receive_Interfaces_Fcn_cEA for IP transports.

- NDDS_Transport_Plugin * **NDDS_Transport_UDPv6_create** (**NDDS_Transport_Address_t** *default_network_address_out, const struct **DDS_PropertyQosPolicy** *property_in)

  Create an instance of a UDPv6 Transport Plugin, using PropertyQosPolicy.

5.67.1 Detailed Description

Built-in transport plug-in using UDP/IPv6 (**NDDS_TRANSPORT_CLASSID_UDPv6** (p. 278)).

This transport plugin uses UDPv6 sockets to send and receive messages. It supports both unicast and multicast communications in a single instance of the plugin. By default, this plugin will use all interfaces that it finds enabled and "UP" at instantiation time to send and receive messages.

The user can configure an instance of this plugin to only use unicast or only use multicast, see **NDDS_Transport_UDPv6_Property_t::unicast_enabled** (p. 1544) and **NDDS_Transport_UDPv6_Property_t::multicast_enabled** (p. 1545).

In addition, the user can configure an instance of this plugin to selectively use the network interfaces of a node (and restrict a plugin from sending multicast messages on specific interfaces) by specifying the "white" and "black" lists in the base property's fields (**NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526), **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526), **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527), **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528)).

RTI Connext can implicitly create this plugin and register it with the **DDSDomainParticipant** (p. 1139) if this transport is specified in the **DDS_TransportBuiltinQosPolicy** (p. 969).

To specify the properties of the builtin UDPv6 transport that is implicitly registered, you can either:

- call **NDDSTransportSupport::set_builtin_transport_property** (p. 1564) or
- specify the predefined property names in **DDS_PropertyQosPolicy** (p. 834) associated with the **DDSDomainParticipant** (p. 1139). (see **UDPv6 Transport Property Names in Property QoS Policy of Domain Participant** (p. 278)). Builtin transport plugin properties specified in **DDS_PropertyQosPolicy** (p. 834) always overwrite the ones specified through **NDDSTransportSupport::set_builtin_transport_property()** (p. 1564). The default value is assumed on any unspecified property.

Note that all properties should be set before the transport is implicitly created and registered by RTI Connext. Any properties that are set after the builtin transport is registered will be ignored. See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

To explicitly create an instance of this plugin, **NDDS_Transport_UDPv6_new()** (p. 280) should be called. The instance should be registered with

RTI Connext, see **NDDSTransportSupport::register_transport** (p. 1560). In some configurations, you may have to disable the builtin UDPv6 transport plugin instance (**DDS_TransportBuiltinQosPolicy** (p. 969), **DDS_TRANSPORTBUILTIN_UDPv6** (p. 398)), to avoid port conflicts with the newly created plugin instance.

5.67.2 UDPv6 Transport Property Names in Property QoS Policy of Domain Participant

The following table lists the predefined property names that can be set in **DDS_PropertyQosPolicy** (p. 834) of a **DDSDomainParticipant** (p. 1139) to configure the builtin UDPv6 transport plugin.

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564)

5.67.3 Define Documentation

5.67.3.1 #define NDDS_TRANSPORT_CLASSID_UDPv6 (5)

Builtin IPv6 UDP/IP Transport-Plugin class ID.

5.67.3.2 #define NDDS_TRANSPORT_UDPV6_ADDRESS_BIT_COUNT (128)

Default value of **NDDS_Transport_Property_t::address_bit_count** (p. 1524).

5.67.3.3 #define NDDS_TRANSPORT_UDPV6_PROPERTIES_BITMAP_DEFAULT (0)

Default value of **NDDS_Transport_Property_t::properties_bitmap** (p. 1524).

5.67.3.4 #define NDDS_TRANSPORT_UDPV6_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT (16)

Default value of **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525).

This is also the maximum value that can be used when instantiating the udp transport.

16 is sufficient for NDDS, but more may improve discovery and reliable performance. Porting note: find out what the maximum gather buffer count is on your OS!

5.67.3.5 #define NDDS_TRANSPORT_UDPV6_SOCKET_BUFFER_SIZE_OS_DEFAULT (-1)

Used to specify that os default be used to specify socket buffer size.

5.67.3.6 #define NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_DEFAULT (9216)

Default value of **NDDS_Transport_Property_t::message_size_max** (p. 1525).

5.67.3.7 #define NDDS_TRANSPORT_UDPV6_MULTICAST_TTL_DEFAULT (1)

Default value of **NDDS_Transport_UDPv6_Property_t::multicast_ttl** (p. 1545).

5.67.3.8 #define NDDS_TRANSPORT_UDPV6_BLOCKING_NEVER

Value for **NDDS_Transport_UDPv6_Property_t::send_blocking** (p. 1547) to specify non-blocking sockets.

5.67.3.9 #define NDDS_TRANSPORT_UDPV6_BLOCKING_ALWAYS

[**default**] Value for **NDDS_Transport_UDPv6_Property_t::send_blocking** (p. 1547) to specify blocking sockets.

5.67.3.10 #define NDDS_TRANSPORT_UDPV6_PROPERTY_DEFAULT

Value:

```
{ \
```

```
{ NDDS_TRANSPORT_CLASSID_UDPv6, \
    NDDS_TRANSPORT_UDPV6_ADDRESS_BIT_COUNT, \
    NDDS_TRANSPORT_UDPV6_PROPERTIES_BITMAP_DEFAULT, \
    NDDS_TRANSPORT_UDPV6_GATHER_SEND_BUFFER_COUNT_MAX_DEFAULT, \
    NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_DEFAULT, \
    NULL, 0, /* allow_interfaces_list */ \
    NULL, 0, /* deny_interfaces_list */ \
    NULL, 0, /* allow_multicast_interfaces_list */ \
    NULL, 0, /* deny_multicast_interfaces_list */ \
}, \
NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_DEFAULT, \
NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_DEFAULT, \
1, /* use unicast */   \
NDDS_TRANSPORT_UDPV6_USE_MULTICAST_DEFAULT, /* use multicast */ \
NDDS_TRANSPORT_UDPV6_MULTICAST_TTL_DEFAULT, \
0, /* multicast loopback enabled */ \
-1, /* (auto-)ignore loopback */ \
0, /* do not ignore non-RUNNING */ \
0, /* no_zero_copy */ \
NDDS_TRANSPORT_UDPV6_BLOCKING_DEFAULT, \
0,  /* enable_v4mapped */ \
0, 0, 0xff /* no mapping to IPV6_TCLASS by default */ }
```

Use this to initialize a **NDDS_Transport_UDPv6_Property_t** (p. 1542) structure.

5.67.4 Function Documentation

5.67.4.1 NDDS_Transport_Plugin* NDDS_Transport_UDPv6_new (const struct NDDS_Transport_UDPv6_Property_t * *property_in*)

Create an instance of a UDPv6 Transport Plugin.

An application may create and register multiple instances of this Transport Plugin with RTI Connext. This may be to partition the network interfaces across multiple RTI Connext domains. However, note that the underlying transport, the operating system's IP layer, is still a "singleton". For example, if a unicast transport has already bound to a port, and another unicast transport tries to bind to the same port, the second attempt will fail.

The transport plugin honors the interface/multicast "white" and "black" lists specified in the **NDDS_Transport_UDPv6_Property_t::parent** (p. 1543):

- **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527),

- **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528)

The format of a string in these lists is assumed to be in standard IPv6 dot notation, possibly containing wildcards. For example:

- 10.10.30.232
- 10.10.*.*
- 192.168.1.*
- etc. Strings not in the correct format will be ignored.

Parameters:

property_in *<<in>>* *(p. 200)* Desired behavior of this transport. May be NULL for default property.

Returns:

A UDPv6 Transport Plugin instance on success; or NULL on failure.

5.67.4.2 RTI_INT32 NDDS_Transport_UDPv6_string_to_address_cEA (NDDS_Transport_Plugin * *self*, NDDS_Transport_Address_t * *address_out*, const char * *address_in*)

Realization of NDDS_Transport_String_To_Address_Fcn_cEA for IP transports.

Converts a host name string to a IPv6 address.

Parameters:

self NOT USED. May be NULL.

address_out *<<out>>* *(p. 200)* The corresponding numerical value in IPv6 format.

address_in *<<in>>* *(p. 200)* The name of the IPv6 address. It can be a dot notation name or a host name. If NULL, then the IP address of the localhost will be returned.

See also:

NDDS_Transport_String_To_Address_Fcn_cEA for complete documentation.

5.67.4.3 RTI_INT32 NDDS_Transport_UDPv6_get_receive_interfaces_cEA (NDDS_Transport_Plugin * *self*, RTI_INT32 * *found_more_than_provided_for_out*, RTI_INT32 * *interface_reported_count_out*, NDDS_Transport_Interface_t *interface_array_inout*[], RTI_INT32 *interface_array_size_in*)

Realization of NDDS_Transport_Get_Receive_Interfaces_Fcn_cEA for IP transports.

Retrieves a list of available IPv6 network interfaces. The addresses returned from IPv6 plugin will be full 128-bit addresses.

See also:

NDDS_Transport_Get_Receive_Interfaces_Fcn_cEA for complete documentation.

5.67.4.4 NDDS_Transport_Plugin* NDDS_Transport_UDPv6_create (NDDS_Transport_Address_t * *default_network_address_out*, const struct DDS_PropertyQosPolicy * *property_in*)

Create an instance of a UDPv6 Transport Plugin, using PropertyQosPolicy.

An application may create and register multiple instances of this Transport Plugin with RTI Connext. This may be to partition the network interfaces across multiple RTI Connext domains. However, note that the underlying transport, the operating system's IP layer, is still a "singleton". For example, if a unicast transport has already bound to a port, and another unicast transport tries to bind to the same port, the second attempt will fail.

The transport plugin honors the interface/multicast "white" and "black" lists specified in the **NDDS_Transport_UDPv6_Property_t::parent** (p. 1543):

- **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526),
- **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527),
- **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528)

The format of a string in these lists is assumed to be in standard IPv6 dot notation, possibly containing wildcards. For example:

- ∗:∗:∗:∗:∗:∗:∗:∗
- FE80:aBc::202:∗:∗
- ∗:aBC::∗2:∗:2∗
- etc. Strings not in the correct format will be ignored.

Parameters:

default_network_address_out <<***out***>> *(*p. *200)* Network address to be used when registering the transport.

property_in <<***in***>> *(*p. *200)* Desired behavior of this transport, through the property field in **DDS_DomainParticipantQos** (p. 588).

Returns:

A UDPv6 Transport Plugin instance on success; or NULL on failure.

Property Name	Description
dds.transport.UDPv6.builtin.parent.address_bit_count	See **NDDS_Transport_Property_t::address_bit_count** (p. 1524)
dds.transport.UDPv6.builtin.parent.properties_bitmap	See **NDDS_Transport_Property_t::properties_bitmap** (p. 1524)
dds.transport.UDPv6.builtin.parent.gather_send_buffer_count_max	See **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525)
dds.transport.UDPv6.builtin.parent.message_size_max	See **NDDS_Transport_Property_t::message_size_max** (p. 1525)
dds.transport.UDPv6.builtin.parent.allow_interfaces	See **NDDS_Transport_Property_t::allow_interfaces_list** (p. 1526) and **NDDS_Transport_Property_t::allow_interfaces_list_length** (p. 1526). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.UDPv6.builtin.parent.deny_interfaces	See **NDDS_Transport_Property_t::deny_interfaces_list** (p. 1526) and **NDDS_Transport_Property_t::deny_interfaces_list_length** (p. 1527). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.UDPv6.builtin.parent.allow_multicast_interfaces	See **NDDS_Transport_Property_t::allow_multicast_interfaces_list** (p. 1527) and **NDDS_Transport_Property_t::allow_multicast_interfaces_list_length** (p. 1528). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0
dds.transport.UDPv6.builtin.parent.deny_multicast_interfaces	See **NDDS_Transport_Property_t::deny_multicast_interfaces_list** (p. 1528) and **NDDS_Transport_Property_t::deny_multicast_interfaces_list_length** (p. 1528). Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example: 127.0.0.1,eth0

5.68 Participant Built-in Topics

Builtin topic for accessing information about the DomainParticipants discovered by RTI Connext.

Classes

- class **DDSParticipantBuiltinTopicDataTypeSupport**

  Instantiates `TypeSupport` < ***DDS_ParticipantBuiltinTopicData*** *(p. 816)* > .

- class **DDSParticipantBuiltinTopicDataDataReader**

  Instantiates `DataReader` < ***DDS_ParticipantBuiltinTopicData*** *(p. 816)* > .

- struct **DDS_ParticipantBuiltinTopicData**

  Entry created when a DomainParticipant object is discovered.

- struct **DDS_ParticipantBuiltinTopicDataSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_ParticipantBuiltinTopicData*** *(p. 816)* > .

Variables

- const char * **DDS_PARTICIPANT_TOPIC_NAME**

  Participant topic name.

5.68.1 Detailed Description

Builtin topic for accessing information about the DomainParticipants discovered by RTI Connext.

5.68.2 Variable Documentation

5.68.2.1 const char* DDS_PARTICIPANT_TOPIC_NAME

Participant topic name.

Topic name of **DDSParticipantBuiltinTopicDataDataReader** (p. 1341)

See also:

DDS_ParticipantBuiltinTopicData (p. 816)
DDSParticipantBuiltinTopicDataDataReader (p. 1341)

5.69 Topic Built-in Topics

Builtin topic for accessing information about the Topics discovered by RTI Connext.

Classes

- class **DDSTopicBuiltinTopicDataTypeSupport**
  Instantiates `TypeSupport` < ***DDS_TopicBuiltinTopicData*** *(p. 958)* > .
- class **DDSTopicBuiltinTopicDataDataReader**
  Instantiates `DataReader` < ***DDS_TopicBuiltinTopicData*** *(p. 958)* > .
- struct **DDS_TopicBuiltinTopicData**
  Entry created when a Topic object discovered.
- struct **DDS_TopicBuiltinTopicDataSeq**
  Instantiates `FooSeq` (p. 1494) < ***DDS_TopicBuiltinTopicData*** *(p. 958)* > .

Variables

- const char * **DDS_TOPIC_TOPIC_NAME**
  Topic topic name.

5.69.1 Detailed Description

Builtin topic for accessing information about the Topics discovered by RTI Connext.

5.69.2 Variable Documentation

5.69.2.1 const char* DDS_TOPIC_TOPIC_NAME

Topic topic name.

Topic name of **DDSTopicBuiltinTopicDataDataReader** (p. 1425)

See also:

DDS_TopicBuiltinTopicData (p. 958)

DDSTopicBuiltinTopicDataDataReader (p. 1425)

5.70 Publication Built-in Topics

Builtin topic for accessing information about the Publications discovered by RTI Connext.

Classes

- class **DDSPublicationBuiltinTopicDataTypeSupport**

  Instantiates `TypeSupport` < ***DDS_PublicationBuiltinTopicData*** *(p. 839)* > .

- class **DDSPublicationBuiltinTopicDataDataReader**

  Instantiates `DataReader` < ***DDS_PublicationBuiltinTopicData*** *(p. 839)* > .

- struct **DDS_PublicationBuiltinTopicData**

  Entry created when a ***DDSDataWriter*** *(p. 1113) is discovered in association with its Publisher.*

- struct **DDS_PublicationBuiltinTopicDataSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_PublicationBuiltinTopicData*** *(p. 839)* > .

Variables

- const char * **DDS_PUBLICATION_TOPIC_NAME**

  Publication topic name.

5.70.1 Detailed Description

Builtin topic for accessing information about the Publications discovered by RTI Connext.

5.70.2 Variable Documentation

5.70.2.1 const char* DDS_PUBLICATION_TOPIC_NAME

Publication topic name.

Topic name of **DDSPublicationBuiltinTopicDataDataReader** (p. 1344)

See also:

DDS_PublicationBuiltinTopicData (p. 839)
DDSPublicationBuiltinTopicDataDataReader (p. 1344)

5.71 Subscription Built-in Topics

Builtin topic for accessing information about the Subscriptions discovered by RTI Connext.

Classes

- class **DDSSubscriptionBuiltinTopicDataTypeSupport**

  Instantiates `TypeSupport` < ***DDS_SubscriptionBuiltinTopicData*** *(p. 936)* > .

- class **DDSSubscriptionBuiltinTopicDataDataReader**

  Instantiates `DataReader` < ***DDS_SubscriptionBuiltinTopicData*** *(p. 936)* > .

- struct **DDS_SubscriptionBuiltinTopicData**

  Entry created when a ***DDSDataReader*** *(p. 1087) is discovered in association with its Subscriber.*

- struct **DDS_SubscriptionBuiltinTopicDataSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_SubscriptionBuiltinTopicData*** *(p. 936)* > .

Variables

- const char * **DDS_SUBSCRIPTION_TOPIC_NAME**

  Subscription topic name.

5.71.1 Detailed Description

Builtin topic for accessing information about the Subscriptions discovered by RTI Connext.

5.71.2 Variable Documentation

5.71.2.1 const char* DDS_SUBSCRIPTION_TOPIC_NAME

Subscription topic name.

Topic name of **DDSSubscriptionBuiltinTopicDataDataReader** (p. 1417)

See also:

DDS_SubscriptionBuiltinTopicData (p. 936)
DDSSubscriptionBuiltinTopicDataDataReader (p. 1417)

5.72 String Built-in Type

Built-in type consisting of a single character string.

Classes

- class **DDSStringTypeSupport**
  <<**interface**>> (p. 199) *String type support.*
- class **DDSStringDataReader**
  <<**interface**>> (p. 199) *Instantiates* `DataReader` < *char** >.
- class **DDSStringDataWriter**
  <<**interface**>> (p. 199) *Instantiates* `DataWriter` < *char** >.

5.72.1 Detailed Description

Built-in type consisting of a single character string.

5.73 KeyedString Built-in Type

Built-in type consisting of a string payload and a second string that is the key.

Classes

- class **DDSKeyedStringTypeSupport**

  <<**interface**>> (p. 199) *Keyed string type support.*

- class **DDSKeyedStringDataReader**

  <<**interface**>> (p. 199) *Instantiates* `DataReader` < ***DDS_KeyedString*** *(p. 768)* >.

- class **DDSKeyedStringDataWriter**

  <<**interface**>> (p. 199) *Instantiates* `DataWriter` < ***DDS_KeyedString*** *(p. 768)* >.

- struct **DDS_KeyedString**

  Keyed string built-in type.

- struct **DDS_KeyedStringSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_KeyedString*** *(p. 768)* > .

5.73.1 Detailed Description

Built-in type consisting of a string payload and a second string that is the key.

5.74 Octets Built-in Type

Built-in type consisting of a variable-length array of opaque bytes.

Classes

- class **DDSOctetsTypeSupport**

  <<**interface**>> (p. 199) ***DDS_Octets*** *(p. 799) type support.*

- class **DDSOctetsDataReader**

  <<**interface**>> (p. 199) *Instantiates* `DataReader` < ***DDS_Octets*** *(p. 799)* >.

- class **DDSOctetsDataWriter**

  <<**interface**>> (p. 199) *Instantiates* `DataWriter` < ***DDS_Octets*** *(p. 799)* >.

- struct **DDS_Octets**

  Built-in type consisting of a variable-length array of opaque bytes.

- struct **DDS_OctetsSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_Octets*** *(p. 799)* > .

5.74.1 Detailed Description

Built-in type consisting of a variable-length array of opaque bytes.

5.75 KeyedOctets Built-in Type

Built-in type consisting of a variable-length array of opaque bytes and a string that is the key.

Classes

- class **DDSKeyedOctetsTypeSupport**

  <<**interface**>> (p. 199) ***DDS_KeyedOctets*** *(p. 765) type support.*

- class **DDSKeyedOctetsDataReader**

  <<**interface**>> (p. 199) *Instantiates* `DataReader` < ***DDS_KeyedOctets*** *(p. 765)* >.

- class **DDSKeyedOctetsDataWriter**

  <<**interface**>> (p. 199) *Instantiates* `DataWriter` < ***DDS_KeyedOctets*** *(p. 765)* >.

- struct **DDS_KeyedOctets**

  Built-in type consisting of a variable-length array of opaque bytes and a string that is the key.

- struct **DDS_KeyedOctetsSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDS_KeyedOctets*** *(p. 765)* >.

5.75.1 Detailed Description

Built-in type consisting of a variable-length array of opaque bytes and a string that is the key.

5.76 DDS-Specific Primitive Types

Basic DDS value types for use in user data types.

Defines

- #define **DDS_BOOLEAN_TRUE**

  Defines "true" value of ***DDS_Boolean*** *(p. 301) data type.*

- #define **DDS_BOOLEAN_FALSE**

  Defines "false" value of ***DDS_Boolean*** *(p. 301) data type.*

Typedefs

- typedef RTICdrChar **DDS_Char**

  Defines a character data type, equivalent to IDL/CDR `char`*.*

- typedef RTICdrWchar **DDS_Wchar**

  Defines a wide character data type, equivalent to IDL/CDR `wchar`*.*

- typedef RTICdrOctet **DDS_Octet**

  Defines an opaque byte data type, equivalent to IDL/CDR `octet`*.*

- typedef RTICdrShort **DDS_Short**

  Defines a short integer data type, equivalent to IDL/CDR `short`*.*

- typedef RTICdrUnsignedShort **DDS_UnsignedShort**

  Defines an unsigned short integer data type, equivalent to IDL/CDR `unsigned short`*.*

- typedef RTICdrLong **DDS_Long**

  Defines a long integer data type, equivalent to IDL/CDR `long`*.*

- typedef RTICdrUnsignedLong **DDS_UnsignedLong**

  Defines an unsigned long integer data type, equivalent to IDL/CDR `unsigned long`*.*

- typedef RTICdrLongLong **DDS_LongLong**

  Defines an extra-long integer data type, equivalent to IDL/CDR `long long`*.*

- typedef RTICdrUnsignedLongLong **DDS_UnsignedLongLong**

Defines an unsigned extra-long data type, equivalent to IDL/CDR `unsigned long long`.

- typedef RTICdrFloat **DDS_Float**

  Defines a single-precision floating-point data type, equivalent to IDL/CDR `float`.

- typedef RTICdrDouble **DDS_Double**

  Defines a double-precision floating-point data type, equivalent to IDL/CDR `double`.

- typedef RTICdrLongDouble **DDS_LongDouble**

  Defines an extra-precision floating-point data type, equivalent to IDL/CDR `long double`.

- typedef RTICdrBoolean **DDS_Boolean**

  Defines a Boolean data type, equivalent to IDL/CDR `boolean`.

- typedef RTICdrEnum **DDS_Enum**

  Defines an enumerated data type.

5.76.1 Detailed Description

Basic DDS value types for use in user data types.

As part of the finalization of the DDS standard, a number of DDS-specific primitive types will be introduced. By using these types, you will ensure that your data is serialized consistently across platforms even if the C/C++ built-in types have different sizes on those platforms.

In this version of RTI Connext, the DDS primitive types are defined using the OMG's Common Data Representation (CDR) standard. In a future version of RTI Connext, you will be given the choice of whether to use these CDR-based types or C/C++ built-in types through a flag provided to the rtiddsgen tool.

Typedef's that begin with RTICdr are defined in <NDDSHOME>/include/ndds/cdr/cdr_type.h, which uses types that are further defined in <NDDSHOME>/include/ndds/osapi/osapi_type.h.

5.76.2 Define Documentation

5.76.2.1 #define DDS_BOOLEAN_TRUE

Defines "true" value of **DDS_Boolean** (p. 301) data type.

5.76.2.2 #define DDS_BOOLEAN_FALSE

Defines "false" value of **DDS_Boolean** (p. 301) data type.

Examples:

HelloWorld.cxx.

5.76.3 Typedef Documentation

5.76.3.1 typedef RTICdrChar DDS_Char

Defines a character data type, equivalent to IDL/CDR `char`.

An 8-bit quantity that encodes a single byte charater from any byte-oriented code set.

5.76.3.2 typedef RTICdrWchar DDS_Wchar

Defines a wide character data type, equivalent to IDL/CDR `wchar`.

An 16-bit quantity that encodes a wide character from any character set.

5.76.3.3 typedef RTICdrOctet DDS_Octet

Defines an opaque byte data type, equivalent to IDL/CDR `octet`.

An 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the middleware.

5.76.3.4 typedef RTICdrShort DDS_Short

Defines a short integer data type, equivalent to IDL/CDR `short`.

A 16-bit signed short integer value.

5.76.3.5 typedef RTICdrUnsignedShort DDS_UnsignedShort

Defines an unsigned short integer data type, equivalent to IDL/CDR `unsigned short`.

A 16-bit unsigned short integer value.

5.76.3.6 typedef RTICdrLong DDS_Long

Defines a long integer data type, equivalent to IDL/CDR `long`.

A 32-bit signed long integer value.

5.76.3.7 typedef RTICdrUnsignedLong DDS_UnsignedLong

Defines an unsigned long integer data type, equivalent to IDL/CDR `unsigned long`.

A 32-bit unsigned long integer value.

5.76.3.8 typedef RTICdrLongLong DDS_LongLong

Defines an extra-long integer data type, equivalent to IDL/CDR `long long`.

A 64-bit signed long long integer value.

5.76.3.9 typedef RTICdrUnsignedLongLong DDS_UnsignedLongLong

Defines an unsigned extra-long data type, equivalent to IDL/CDR `unsigned long long`.

An 64-bit unsigned long long integer value.

5.76.3.10 typedef RTICdrFloat DDS_Float

Defines a single-precision floating-point data type, equivalent to IDL/CDR `float`.

A 32-bit floating-point value.

5.76.3.11 typedef RTICdrDouble DDS_Double

Defines a double-precision floating-point data type, equivalent to IDL/CDR `double`.

A 64-bit floating-point value.

5.76.3.12 typedef RTICdrLongDouble DDS_LongDouble

Defines an extra-precision floating-point data type, equivalent to IDL/CDR `long double`.

A 128-bit floating-point value.

Since some architectures do not support long double, RTI has defined character arrays that match the expected size of this type. On systems that do have native long double, you have to define RTI_CDR_SIZEOF_LONG_DOUBLE as 16 to map them to native types.

5.76.3.13 typedef RTICdrBoolean DDS_Boolean

Defines a Boolean data type, equivalent to IDL/CDR `boolean`.

An 8-bit Boolean value that is used to denote a data item that can only take one of the values **DDS_BOOLEAN_TRUE** (p. 298) (1) or **DDS_BOOLEAN_FALSE** (p. 299) (0).

5.76.3.14 typedef RTICdrEnum DDS_Enum

Defines an enumerated data type.

Encoded as unsigned long value. By default, the first enum identifier has the numeric value zero (0). Successive enum identifiers take ascending numeric values, in order of declaration from left to right.

5.77 Time Support

Time and duration types and defines.

Classes

- struct **DDS_Time_t**

  Type for time *representation.*

- struct **DDS_Duration_t**

  Type for duration *representation.*

Defines

- #define **DDS_TIME_ZERO**

  The default instant in time: zero seconds and zero nanoseconds.

Functions

- **DDS_Boolean DDS_Time_is_zero** (const struct **DDS_Time_t** *time)

  Check if time is zero.

- **DDS_Boolean DDS_Time_is_invalid** (const struct **DDS_Time_t** *time)
- **DDS_Boolean DDS_Duration_is_infinite** (const struct **DDS_Duration_t** *duration)
- **DDS_Boolean DDS_Duration_is_auto** (const struct **DDS_Duration_t** *duration)
- **DDS_Boolean DDS_Duration_is_zero** (const struct **DDS_Duration_t** *duration)

Variables

- const **DDS_Long DDS_TIME_INVALID_SEC**

  A sentinel indicating an invalid second of time.

- const **DDS_UnsignedLong DDS_TIME_INVALID_NSEC**

  A sentinel indicating an invalid nano-second of time.

- struct **DDS_Time_t DDS_TIME_INVALID**
  A sentinel indicating an invalid time.
- const **DDS_Long DDS_DURATION_INFINITE_SEC**
  An infinite second period of time.
- const **DDS_UnsignedLong DDS_DURATION_INFINITE_NSEC**
  An infinite nano-second period of time.
- struct **DDS_Duration_t DDS_DURATION_INFINITE**
  An infinite period of time.
- const **DDS_Long DDS_DURATION_AUTO_SEC**
  An auto second period of time.
- const **DDS_UnsignedLong DDS_DURATION_AUTO_NSEC**
  An auto nano-second period of time.
- struct **DDS_Duration_t DDS_DURATION_AUTO**
  Duration is automatically assigned.
- const **DDS_Long DDS_DURATION_ZERO_SEC**
  A zero-length second period of time.
- const **DDS_UnsignedLong DDS_DURATION_ZERO_NSEC**
  A zero-length nano-second period of time.
- struct **DDS_Duration_t DDS_DURATION_ZERO**
  A zero-length period of time.

5.77.1 Detailed Description

Time and duration types and defines.

5.77.2 Define Documentation

5.77.2.1 #define DDS_TIME_ZERO

The default instant in time: zero seconds and zero nanoseconds.

5.77.3 Function Documentation

5.77.3.1 DDS_Boolean DDS_Time_is_zero (const struct DDS_Time_t * *time*)

Check if time is zero.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given time is equal to **DDS_TIME_ZERO** (p. 303) or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

5.77.3.2 DDS_Boolean DDS_Time_is_invalid (const struct DDS_Time_t * *time*)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given time is not valid (i.e. is negative)

5.77.3.3 DDS_Boolean DDS_Duration_is_infinite (const struct DDS_Duration_t * *duration*)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given duration is of infinite length.

5.77.3.4 DDS_Boolean DDS_Duration_is_auto (const struct DDS_Duration_t * *duration*)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given duration has auto value.

5.77.3.5 DDS_Boolean DDS_Duration_is_zero (const struct DDS_Duration_t * *duration*)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the given duration is of zero length.

5.77.4 Variable Documentation

5.77.4.1 const DDS_Long DDS_TIME_INVALID_SEC

A sentinel indicating an invalid second of time.

5.77.4.2 const DDS_UnsignedLong DDS_TIME_INVALID_NSEC

A sentinel indicating an invalid nano-second of time.

5.77.4.3 struct DDS_Time_t DDS_TIME_INVALID

A sentinel indicating an invalid time.

5.77.4.4 const DDS_Long DDS_DURATION_INFINITE_SEC

An infinite second period of time.

5.77.4.5 const DDS_UnsignedLong DDS_DURATION_INFINITE_NSEC

An infinite nano-second period of time.

5.77.4.6 struct DDS_Duration_t DDS_DURATION_INFINITE

An infinite period of time.

5.77.4.7 const DDS_Long DDS_DURATION_AUTO_SEC

An auto second period of time.

5.77.4.8 const DDS_UnsignedLong DDS_DURATION_AUTO_NSEC

An auto nano-second period of time.

5.77.4.9 struct DDS_Duration_t DDS_DURATION_AUTO

Duration is automatically assigned.

5.77.4.10 const DDS_Long DDS_DURATION_ZERO_SEC

A zero-length second period of time.

5.77.4.11 const DDS_UnsignedLong DDS_DURATION_ZERO_NSEC

A zero-length nano-second period of time.

5.77.4.12 struct DDS_Duration_t DDS_DURATION_ZERO

A zero-length period of time.

5.78 GUID Support

<<*eXtension*>> *(*p. *199)* GUID type and defines.

Classes

- struct **DDS_GUID_t**

  Type for GUID *(Global Unique Identifier) representation.*

Functions

- **DDS_Boolean DDS_GUID_equals** (const struct **DDS_GUID_t** *self, const struct **DDS_GUID_t** *other)

  Compares this GUID with another GUID for equality.

- int **DDS_GUID_compare** (const struct **DDS_GUID_t** *self, const struct **DDS_GUID_t** *other)

  Compares two GUIDs.

- void **DDS_GUID_copy** (struct **DDS_GUID_t** *self, const struct **DDS_GUID_t** *other)

  Copies another GUID into this GUID.

Variables

- struct **DDS_GUID_t DDS_GUID_AUTO**

  Indicates that RTI Connext should choose an appropriate virtual GUID.

- struct **DDS_GUID_t DDS_GUID_UNKNOWN**

  Unknown GUID.

5.78.1 Detailed Description

<<*eXtension*>> *(*p. *199)* GUID type and defines.

5.78.2 Function Documentation

5.78.2.1 DDS_Boolean DDS_GUID_equals (const struct DDS_GUID_t * *self*, const struct DDS_GUID_t * *other*)

Compares this GUID with another GUID for equality.

Parameters:

self *<<in>>* *(p. 200)* This GUID. Cannot be NULL.

other *<<in>>* *(p. 200)* The other GUID to be compared with this GUID. Cannot be NULL.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the two GUIDs have equal values, or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

5.78.2.2 int DDS_GUID_compare (const struct DDS_GUID_t * *self*, const struct DDS_GUID_t * *other*)

Compares two GUIDs.

Parameters:

self *<<in>>* *(p. 200)* GUID to compare. Cannot be NULL.

other *<<in>>* *(p. 200)* GUID to compare. Cannot be NULL.

Returns:

If the two GUIDs are equal, the function returns 0. If self is greater than other the function returns a positive number; otherwise, it returns a negative number.

5.78.2.3 void DDS_GUID_copy (struct DDS_GUID_t * *self*, const struct DDS_GUID_t * *other*)

Copies another GUID into this GUID.

Parameters:

self *<<in>>* *(p. 200)* This GUID. Cannot be NULL.

other *<<in>>* *(p. 200)* The other GUID to be copied.

5.78.3 Variable Documentation

5.78.3.1 struct DDS_GUID_t DDS_GUID_AUTO

Indicates that RTI Connext should choose an appropriate virtual GUID.

If this special value is assigned to **DDS_DataWriterProtocolQosPolicy::virtual_guid** (p. 536) or **DDS_DataReaderProtocolQosPolicy::virtual_guid** (p. 502), RTI Connext will assign the virtual GUID automatically based on the RTPS or physical GUID.

5.78.3.2 struct DDS_GUID_t DDS_GUID_UNKNOWN

Unknown GUID.

5.79 Sequence Number Support

<<*eXtension*>> *(*p. *199)* Sequence number type and defines.

Classes

- struct **DDS_SequenceNumber_t**

  Type for sequence *number representation.*

Variables

- struct **DDS_SequenceNumber_t DDS_SEQUENCE_NUMBER_UNKNOWN**

  Unknown sequence number.

- struct **DDS_SequenceNumber_t DDS_SEQUENCE_NUMBER_ZERO**

  Zero value for the sequence number.

- struct **DDS_SequenceNumber_t DDS_SEQUENCE_NUMBER_MAX**

  Highest, most positive value for the sequence number.

- struct **DDS_SequenceNumber_t DDS_AUTO_SEQUENCE_NUMBER**

  The sequence number is internally determined by RTI Connext.

5.79.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Sequence number type and defines.

5.79.2 Variable Documentation

5.79.2.1 struct DDS_SequenceNumber_t DDS_SEQUENCE_NUMBER_UNKNOWN

Unknown sequence number.

5.79.2.2 struct DDS_SequenceNumber_t DDS_SEQUENCE_NUMBER_ZERO

Zero value for the sequence number.

5.79.2.3 struct DDS_SequenceNumber_t DDS_SEQUENCE_NUMBER_MAX

Highest, most positive value for the sequence number.

5.79.2.4 struct DDS_SequenceNumber_t DDS_AUTO_SEQUENCE_NUMBER

The sequence number is internally determined by RTI Connext.

5.80 Exception Codes

<<*eXtension*>> *(*p. *199)* Exception codes.

Enumerations

ˆ enum **DDS_ExceptionCode_t** {
DDS_NO_EXCEPTION_CODE,
DDS_USER_EXCEPTION_CODE,
DDS_SYSTEM_EXCEPTION_CODE,
DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE,
DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE,
DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE,
DDS_BADKIND_USER_EXCEPTION_CODE,
DDS_BOUNDS_USER_EXCEPTION_CODE,
DDS_IMMUTABLE_TYPECODE_SYSTEM_EXCEPTION_CODE = 8,
DDS_BAD_MEMBER_NAME_USER_EXCEPTION_CODE = 9,
DDS_BAD_MEMBER_ID_USER_EXCEPTION_CODE = 10 }
Error codes used by the ***DDS_TypeCode*** *(*p. *992) class.*

5.80.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Exception codes.

These exceptions are used for error handling by the **Type Code Support** (p. 56) API.

5.80.2 Enumeration Type Documentation

5.80.2.1 enum DDS_ExceptionCode_t

Error codes used by the **DDS_TypeCode** (p. 992) class.

Exceptions are modeled via a special parameter passed to the operations.

Enumerator:

DDS_NO_EXCEPTION_CODE No failure occurred.

DDS_USER_EXCEPTION_CODE User exception.

This class is based on a similar class in CORBA.

DDS_SYSTEM_EXCEPTION_CODE System exception.

This class is based on a similar class in CORBA.

DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE Exception thrown when a parameter passed to a call is considered illegal.

DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE Exception thrown when there is not enough memory for a dynamic memory allocation.

DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE Exception thrown when a malformed type code is found (for example, a type code with an invalid TCKind value).

DDS_BADKIND_USER_EXCEPTION_CODE The exception BadKind is thrown when an inappropriate operation is invoked on a TypeCode object.

DDS_BOUNDS_USER_EXCEPTION_CODE A user exception thrown when a parameter is not within the legal bounds.

DDS_IMMUTABLE_TYPECODE_SYSTEM_EXCEPTION_CODE An attempt was made to modify a **DDS_TypeCode** (p. 992) that was received from a remote object.

The built-in publication and subscription readers provide access to information about the remote **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) entities in the distributed system. Among other things, the data from these built-in readers contains the **DDS_TypeCode** (p. 992) for these entities. Modifying this received **DDS_TypeCode** (p. 992) is not permitted.

DDS_BAD_MEMBER_NAME_USER_EXCEPTION_CODE The specified **DDS_TypeCode** (p. 992) member name is invalid.

This failure can occur, for example, when querying a field by name when no such name is defined in the type.

See also:

DDS_BAD_MEMBER_ID_USER_EXCEPTION_CODE (p. 313)

DDS_BAD_MEMBER_ID_USER_EXCEPTION_CODE The specified **DDS_TypeCode** (p. 992) member ID is invalid.

This failure can occur, for example, when querying a field by ID when no such ID is defined in the type.

See also:

DDS_BAD_MEMBER_NAME_USER_EXCEPTION_CODE (p. 313)

5.81 Return Codes

Types of return codes.

Enumerations

ˆ enum **DDS_ReturnCode_t** {
DDS_RETCODE_OK,
DDS_RETCODE_ERROR,
DDS_RETCODE_UNSUPPORTED,
DDS_RETCODE_BAD_PARAMETER,
DDS_RETCODE_PRECONDITION_NOT_MET,
DDS_RETCODE_OUT_OF_RESOURCES,
DDS_RETCODE_NOT_ENABLED,
DDS_RETCODE_IMMUTABLE_POLICY,
DDS_RETCODE_INCONSISTENT_POLICY,
DDS_RETCODE_ALREADY_DELETED,
DDS_RETCODE_TIMEOUT,
DDS_RETCODE_NO_DATA,
DDS_RETCODE_ILLEGAL_OPERATION }
Type for return codes.

5.81.1 Detailed Description

Types of return codes.

5.81.2 Standard Return Codes

Any operation with return type **DDS_ReturnCode_t** (p. 315) may return **DDS_RETCODE_OK** (p. 315) **DDS_RETCODE_ERROR** (p. 315) or **DDS_RETCODE_ILLEGAL_OPERATION** (p. 316). Any operation that takes one or more input parameters may additionally return **DDS_RETCODE_BAD_PARAMETER** (p. 315). Any operation on an object created from any of the factories may additionally return **DDS_RETCODE_ALREADY_DELETED** (p. 316). Any operation that is stated as optional may additionally return **DDS_RETCODE_UNSUPPORTED** (p. 315).

Thus, the standard return codes are:

- **DDS_RETCODE_ERROR** (p. 315)
- **DDS_RETCODE_ILLEGAL_OPERATION** (p. 316)
- **DDS_RETCODE_ALREADY_DELETED** (p. 316)
- **DDS_RETCODE_BAD_PARAMETER** (p. 315)
- **DDS_RETCODE_UNSUPPORTED** (p. 315)

Operations that may return any of the additional return codes will state so explicitly.

5.81.3 Enumeration Type Documentation

5.81.3.1 enum DDS_ReturnCode_t

Type for return codes.

Errors are modeled as operation return codes of this type.

Enumerator:

DDS_RETCODE_OK Successful return.

DDS_RETCODE_ERROR Generic, unspecified error.

DDS_RETCODE_UNSUPPORTED Unsupported operation. Can only returned by operations that are unsupported.

DDS_RETCODE_BAD_PARAMETER Illegal parameter value.

The value of the parameter that is passed in has llegal value. Things that falls into this category includes NULL parameters and parameter values that are out of range.

DDS_RETCODE_PRECONDITION_NOT_MET A pre-condition for the operation was not met.

The system is not in the expected state when the function is called, or the parameter itself is not in the expected state when the function is called.

DDS_RETCODE_OUT_OF_RESOURCES RTI Connext ran out of the resources needed to complete the operation.

DDS_RETCODE_NOT_ENABLED Operation invoked on a **DDSEntity** (p. 1253) that is not yet enabled.

DDS_RETCODE_IMMUTABLE_POLICY Application attempted to modify an immutable QoS policy.

DDS_RETCODE_INCONSISTENT_POLICY Application specified a set of QoS policies that are not consistent with each other.

DDS_RETCODE_ALREADY_DELETED The object target of this operation has already been deleted.

DDS_RETCODE_TIMEOUT The operation timed out.

DDS_RETCODE_NO_DATA Indicates a transient situation where the operation did not return any data but there is no inherent error.

DDS_RETCODE_ILLEGAL_OPERATION The operation was called under improper circumstances.

An operation was invoked on an inappropriate object or at an inappropriate time. This return code is similar to **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), except that there is no precondition that could be changed to make the operation succeed.

5.82 Status Kinds

Kinds of communication status.

Defines

- #define **DDS_STATUS_MASK_NONE**
  No bits are set.
- #define **DDS_STATUS_MASK_ALL**
  All bits are set.

Typedefs

- typedef **DDS_UnsignedLong DDS_StatusMask**
  A bit-mask (list) of concrete status types, i.e. ***DDS_StatusKind*** *(*p. 322*)[].*

Enumerations

- enum **DDS_StatusKind** {
  DDS_INCONSISTENT_TOPIC_STATUS,
  DDS_OFFERED_DEADLINE_MISSED_STATUS,
  DDS_REQUESTED_DEADLINE_MISSED_STATUS,
  DDS_OFFERED_INCOMPATIBLE_QOS_STATUS,
  DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS,
  DDS_SAMPLE_LOST_STATUS,
  DDS_SAMPLE_REJECTED_STATUS,
  DDS_DATA_ON_READERS_STATUS,
  DDS_DATA_AVAILABLE_STATUS,
  DDS_LIVELINESS_LOST_STATUS,
  DDS_LIVELINESS_CHANGED_STATUS,
  DDS_PUBLICATION_MATCHED_STATUS,
  DDS_SUBSCRIPTION_MATCHED_STATUS ,
  DDS_DATA_WRITER_INSTANCE_REPLACED_STATUS,
  DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS,
  DDS_RELIABLE_READER_ACTIVITY_CHANGED_STATUS,

DDS_DATA_WRITER_CACHE_STATUS,

DDS_DATA_WRITER_PROTOCOL_STATUS,

DDS_DATA_READER_CACHE_STATUS,

DDS_DATA_READER_PROTOCOL_STATUS }

Type for status *kinds.*

5.82.1 Detailed Description

Kinds of communication status.

Entity:

DDSEntity (p. 1253)

QoS:

QoS Policies (p. 331)

Listener:

DDSListener (p. 1318)

Each concrete **DDSEntity** (p. 1253) is associated with a set of Status objects whose value represents the communication status of that entity. Each status value can be accessed with a corresponding method on the **DDSEntity** (p. 1253).

When these status values change, the corresponding **DDSStatusCondition** (p. 1376) objects are activated and the proper **DDSListener** (p. 1318) objects are invoked to asynchronously inform the application.

An application is notified of communication status by means of the **DDSListener** (p. 1318) or the **DDSWaitSet** (p. 1433) / **DDSCondition** (p. 1075) mechanism. The two mechanisms may be combined in the application (e.g., using **DDSWaitSet** (p. 1433) (s) / **DDSCondition** (p. 1075) (s) to access the data and **DDSListener** (p. 1318) (s) to be warned asynchronously of erroneous communication statuses).

It is likely that the application will choose one or the other mechanism for each particular communication status (not both). However, if both mechanisms are enabled, then the **DDSListener** (p. 1318) mechanism is used first and then the **DDSWaitSet** (p. 1433) objects are signalled.

The statuses may be classified into:

- *read communication statuses*: i.e., those that are related to arrival of data, namely **DDS_DATA_ON_READERS_STATUS** (p. 324) and **DDS_DATA_AVAILABLE_STATUS** (p. 324).

- *plain communication statuses*: i.e., all the others.

Read communication statuses are treated slightly differently than the others because they don't change independently. In other words, at least two changes will appear at the same time (**DDS_DATA_ON_READERS_STATUS** (p. 324) and **DDS_DATA_AVAILABLE_STATUS** (p. 324)) and even several of the last kind may be part of the set. This 'grouping' has to be communicated to the application.

For each plain communication status, there is a corresponding structure to hold the status value. These values contain the information related to the change of status, as well as information related to the statuses themselves (e.g., contains cumulative counts).

5.82.2 Changes in Status

Associated with each one of an **DDSEntity** (p. 1253)'s communication status is a logical `StatusChangedFlag`. This flag indicates whether that particular communication status has changed since the last time the status was read by the application. The way the status changes is slightly different for the Plain Communication Status and the Read Communication status.

5.82.2.1 Changes in plain communication status

For the plain communication status, the `StatusChangedFlag` flag is initially set to FALSE. It becomes TRUE whenever the plain communication status changes and it is reset to **DDS_BOOLEAN_FALSE** (p. 299) each time the application accesses the plain communication status via the proper `get_<plain communication status>()` operation on the **DDSEntity** (p. 1253).

The communication status is also reset to FALSE whenever the associated listener operation is called as the listener implicitly accesses the status which is passed as a parameter to the operation. The fact that the status is reset prior to calling the listener means that if the application calls the get_<plain communication status> from inside the listener it will see the status already reset.

An exception to this rule is when the associated listener is the 'nil' listener. The 'nil' listener is treated as a NOOP and the act of calling the 'nil' listener does not reset the communication status.

For example, the value of the `StatusChangedFlag` associated with the **DDS_REQUESTED_DEADLINE_MISSED_STATUS** (p. 323) will be-

come TRUE each time new deadline occurs (which increases the **DDS_RequestedDeadlineMissedStatus::total_count** (p. 875) field). The value changes to FALSE when the application accesses the status via the corresponding **DDSDataReader::get_requested_deadline_missed_status** (p. 1099) method on the proper Entity

5.82.2.2 Changes in read communication status

For the read communication status, the `StatusChangedFlag` flag is initially set to FALSE. The `StatusChangedFlag` becomes TRUE when either a data-sample arrives or else the **DDS_ViewStateKind** (p. 113), **DDS_SampleStateKind** (p. 111), or **DDS_InstanceStateKind** (p. 116) of any existing sample changes for any reason other than a call to **FooDataReader::read** (p. 1447), **FooDataReader::take** (p. 1448) or their variants. Specifically any of the following events will cause the `StatusChangedFlag` to become TRUE:

- The arrival of new data.
- A change in the **DDS_InstanceStateKind** (p. 116) of a contained instance. This can be caused by either:
  - The arrival of the notification that an instance has been disposed by:
    - the **DDSDataWriter** (p. 1113) that owns it if **OWNERSHIP** (p. 355) QoS kind= **DDS_EXCLUSIVE_OWNERSHIP_QOS** (p. 356)
    - or by any **DDSDataWriter** (p. 1113) if **OWNERSHIP** (p. 355) QoS kind= **DDS_SHARED_OWNERSHIP_QOS** (p. 356)
  - The loss of liveliness of the **DDSDataWriter** (p. 1113) of an instance for which there is no other **DDSDataWriter** (p. 1113).
  - The arrival of the notification that an instance has been unregistered by the only **DDSDataWriter** (p. 1113) that is known to be writing the instance.

Depending on the kind of `StatusChangedFlag`, the flag transitions to FALSE again as follows:

- The **DDS_DATA_AVAILABLE_STATUS** (p. 324) `StatusChangedFlag` becomes FALSE when either the corresponding listener operation (on_data_available) is called or the read or take operation (or their variants) is called on the associated **DDSDataReader** (p. 1087).
- The **DDS_DATA_ON_READERS_STATUS** (p. 324) `StatusChangedFlag` becomes FALSE when any of the following events occurs:

- The corresponding listener operation (on_data_on_readers) is called.
- The on_data_available listener operation is called on any **DDSDataReader** (p. 1087) belonging to the **DDSSubscriber** (p. 1390).
- The read or take operation (or their variants) is called on any **DDSDataReader** (p. 1087) belonging to the **DDSSubscriber** (p. 1390).

See also:

DDSListener (p. 1318)
DDSWaitSet (p. 1433), **DDSCondition** (p. 1075)

5.82.3 Define Documentation

5.82.3.1 #define DDS_STATUS_MASK_NONE

No bits are set.

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

5.82.3.2 #define DDS_STATUS_MASK_ALL

All bits are set.

Examples:

HelloWorld_subscriber.cxx.

5.82.4 Typedef Documentation

5.82.4.1 typedef DDS_UnsignedLong DDS_StatusMask

A bit-mask (list) of concrete status types, i.e. **DDS_StatusKind** (p. 322)[].

The bit-mask is an efficient and compact representation of a fixed-length list of **DDS_StatusKind** (p. 322) values.

Bits in the mask correspond to different statuses. You can choose which changes in status will trigger a callback by setting the corresponding status bits in this bit-mask and installing callbacks for each of those statuses.

The bits that are true indicate that the listener will be called back for changes in the corresponding status.

For example:

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
                                    DDS_DATA_AVAILABLE_STATUS;
                datareader->set_listener(listener, mask);
```

or

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
                                    DDS_DATA_AVAILABLE_STATUS;
                datareader = subscriber->create_datareader(topic,
                                                DDS_DATAREADER_QOS_DEFAULT,
                                                listener, mask);
```

5.82.5 Enumeration Type Documentation

5.82.5.1 enum DDS_StatusKind

Type for *status* kinds.

Each concrete **DDSEntity** (p. 1253) is associated with a set of ∗`Status` objects whose values represent the communication status of that **DDSEntity** (p. 1253).

The communication statuses whose changes can be communicated to the application depend on the **DDSEntity** (p. 1253).

Each status value can be accessed with a corresponding method on the **DDSEntity** (p. 1253). The changes on these status values cause activation of the corresponding **DDSStatusCondition** (p. 1376) objects and trigger invocation of the proper **DDSListener** (p. 1318) objects to asynchronously inform the application.

See also:

DDSEntity (p. 1253), **DDSStatusCondition** (p. 1376), **DDSListener** (p. 1318)

Enumerator:

DDS_INCONSISTENT_TOPIC_STATUS Another topic exists with the same name but different characteristics.

Entity:

DDSTopic (p. 1419)

Status:

DDS_InconsistentTopicStatus (p. 762)

Listener:

DDSTopicListener (p. 1430)

DDS_OFFERED_DEADLINE_MISSED_STATUS The deadline that the **DDSDataWriter** (p. 1113) has committed through its **DDS_DeadlineQosPolicy** (p. 567) was not respected for a specific instance.
Entity:

DDSDataWriter (p. 1113)
QoS:

DEADLINE (p. 353)
Status:

DDS_OfferedDeadlineMissedStatus (p. 803)
Listener:

DDSDataWriterListener (p. 1133)

DDS_REQUESTED_DEADLINE_MISSED_STATUS The deadline that the **DDSDataReader** (p. 1087) was expecting through its **DDS_DeadlineQosPolicy** (p. 567) was not respected for a specific instance.
Entity:

DDSDataReader (p. 1087)
QoS:

DEADLINE (p. 353)
Status:

DDS_RequestedDeadlineMissedStatus (p. 875)
Listener:

DDSDataReaderListener (p. 1108)

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS A QosPolicy value was incompatible with what was requested.
Entity:

DDSDataWriter (p. 1113)
Status:

DDS_OfferedIncompatibleQosStatus (p. 805)
Listener:

DDSDataWriterListener (p. 1133)

DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS A QosPolicy value was incompatible with what is offered.
Entity:

DDSDataReader (p. 1087)

Status:

DDS_RequestedIncompatibleQosStatus (p. 877)

Listener:

DDSDataReaderListener (p. 1108)

DDS_SAMPLE_LOST_STATUS A sample has been lost (i.e. was never received).

Entity:

DDSDataReader (p. 1087)

Status:

DDS_SampleLostStatus (p. 923)

Listener:

DDSDataReaderListener (p. 1108)

DDS_SAMPLE_REJECTED_STATUS A (received) sample has been rejected.

Entity:

DDSDataReader (p. 1087)

QoS:

RESOURCE_LIMITS (p. 371)

Status:

DDS_SampleRejectedStatus (p. 925)

Listener:

DDSDataReaderListener (p. 1108)

DDS_DATA_ON_READERS_STATUS New data is available.

Entity:

DDSSubscriber (p. 1390)

Listener:

DDSSubscriberListener (p. 1414)

DDS_DATA_AVAILABLE_STATUS One or more new data samples have been received.

Entity:

DDSDataReader (p. 1087)

Listener:

DDSDataReaderListener (p. 1108)

DDS_LIVELINESS_LOST_STATUS The liveliness that the **DDSDataWriter** (p. 1113) has committed to through its **DDS_LivelinessQosPolicy** (p. 779) was not respected, thus **DDSDataReader** (p. 1087) entities will consider the **DDSDataWriter** (p. 1113) as no longer alive.

Entity:

DDSDataWriter (p. 1113)

QoS:

LIVELINESS (p. 358)

Status:

DDS_LivelinessLostStatus (p. 777)

Listener:

DDSDataWriterListener (p. 1133)

DDS_LIVELINESS_CHANGED_STATUS The liveliness of one or more **DDSDataWriter** (p. 1113) that were writing instances read through the **DDSDataReader** (p. 1087) has changed. Some **DDSDataWriter** (p. 1113) have become alive or not_alive.

Entity:

DDSDataReader (p. 1087)

QoS:

LIVELINESS (p. 358)

Status:

DDS_LivelinessChangedStatus (p. 775)

Listener:

DDSDataReaderListener (p. 1108)

DDS_PUBLICATION_MATCHED_STATUS The **DDSDataWriter** (p. 1113) has found **DDSDataReader** (p. 1087) that matches the **DDSTopic** (p. 1419) and has compatible QoS.

Entity:

DDSDataWriter (p. 1113)

Status:

DDS_PublicationMatchedStatus (p. 848)

Listener:

DDSDataWriterListener (p. 1133)

DDS_SUBSCRIPTION_MATCHED_STATUS The **DDSDataReader** (p. 1087) has found **DDSDataWriter** (p. 1113) that matches the **DDSTopic** (p. 1419) and has compatible QoS.

Entity:

DDSDataReader (p. 1087)

Status:

DDS_SubscriptionMatchedStatus (p. 945)

Listener:

DDSDataReaderListener (p. 1108)

DDS_DATA_WRITER_INSTANCE_REPLACED_STATUS
<<***eXtension***>> *(p. 199)* A **DDSDataWriter** (p. 1113) instance has been replaced

Enables a **DDSDataWriter** (p. 1113) callback that is called when an instance in the writer queue is replaced.

Entity:

DDSDataWriter (p. 1113)

Listener:

DDSDataWriterListener (p. 1133)

DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS
<<***eXtension***>> *(p. 199)* The number of unacknowledged samples in a reliable writer's cache has changed such that it has reached a pre-defined trigger point.

This status is considered changed at the following times: the cache is empty (i.e. contains no unacknowledge samples), full (i.e. the sample count has reached the value specified in **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881)), or the number of samples has reached a high (see **DDS_RtpsReliableWriterProtocol_t::high_watermark** (p. 892)) or low (see **DDS_RtpsReliableWriterProtocol_t::low_watermark** (p. 892)) watermark.

Entity:

DDSDataWriter (p. 1113)

Status:

DDS_ReliableWriterCacheChangedStatus (p. 871)

Listener:

DDSDataWriterListener (p. 1133)

DDS_RELIABLE_READER_ACTIVITY_CHANGED_STATUS *<<eXtension>>* *(p. 199)* One or more reliable readers has become active or inactive.

A reliable reader is considered active by a reliable writer with which it is matched if that reader acknowledges the samples it has been sent in a timely fashion. For the definition of "timely" in this case, see **DDS_RtpsReliableWriterProtocol_t** (p. 889) and **DDS_ReliableReaderActivityChangedStatus** (p. 869).

See also:

DDS_RtpsReliableWriterProtocol_t (p. 889)
DDS_ReliableReaderActivityChangedStatus (p. 869)

DDS_DATA_WRITER_CACHE_STATUS *<<eXtension>>* *(p. 199)* The status of the writer's cache.

DDS_DATA_WRITER_PROTOCOL_STATUS *<<eXtension>>* *(p. 199)* The status of a writer's internal protocol related metrics

The status of a writer's internal protocol related metrics, like the number of samples pushed, pulled, filtered; and status of wire protocol traffic.

DDS_DATA_READER_CACHE_STATUS *<<eXtension>>* *(p. 199)* The status of the reader's cache.

DDS_DATA_READER_PROTOCOL_STATUS *<<eXtension>>* *(p. 199)* The status of a reader's internal protocol related metrics

The status of a reader's internal protocol related metrics, like the number of samples received, filtered, rejected; and status of wire protocol traffic.

5.83 Thread Settings

The properties of a thread of execution.

Classes

- struct **DDS_ThreadSettings_t**

  The properties of a thread of execution.

Defines

- #define **DDS_THREAD_SETTINGS_KIND_MASK_DEFAULT**

  The mask of default thread options.

Typedefs

- typedef **DDS_UnsignedLong DDS_ThreadSettingsKindMask**

  *A mask of which each bit is taken from **DDS_ThreadSettingsKind** (p. 329).*

Enumerations

- enum **DDS_ThreadSettingsKind** {
  DDS_THREAD_SETTINGS_FLOATING_POINT,
  DDS_THREAD_SETTINGS_STDIO,
  DDS_THREAD_SETTINGS_REALTIME_PRIORITY,
  DDS_THREAD_SETTINGS_PRIORITY_ENFORCE }

  A collection of flags used to configure threads of execution.

- enum **DDS_ThreadSettingsCpuRotationKind** {
  DDS_THREAD_SETTINGS_CPU_NO_ROTATION,
  DDS_THREAD_SETTINGS_CPU_RR_ROTATION }

  *Determines how **DDS_ThreadSettings_t::cpu_list** (p. 951) affects processor affinity for thread-related QoS policies that apply to multiple threads.*

5.83.1 Detailed Description

The properties of a thread of execution.

5.83.2 Define Documentation

5.83.2.1 #define DDS_THREAD_SETTINGS_KIND_MASK_DEFAULT

The mask of default thread options.

5.83.3 Typedef Documentation

5.83.3.1 typedef DDS_UnsignedLong DDS_ThreadSettingsKindMask

A mask of which each bit is taken from **DDS_ThreadSettingsKind** (p. 329).

See also:

DDS_ThreadSettings_t (p. 950)

5.83.4 Enumeration Type Documentation

5.83.4.1 enum DDS_ThreadSettingsKind

A collection of flags used to configure threads of execution.

Not all of these options may be relevant for all operating systems.

See also:

DDS_ThreadSettingsKindMask (p. 329)

Enumerator:

DDS_THREAD_SETTINGS_FLOATING_POINT Code executed within the thread may perform floating point operations.

DDS_THREAD_SETTINGS_STDIO Code executed within the thread may access standard I/O.

DDS_THREAD_SETTINGS_REALTIME_PRIORITY The thread will be schedule on a real-time basis.

DDS_THREAD_SETTINGS_PRIORITY_ENFORCE Strictly enforce this thread's priority.

5.83.4.2 enum DDS_ThreadSettingsCpuRotationKind

Determines how **DDS_ThreadSettings_t::cpu_list** (p. 951) affects processor affinity for thread-related QoS policies that apply to multiple threads.

5.83.5 Controlling CPU Core Affinity for RTI Threads

Most thread-related QoS settings apply to a single thread (such as for the **DDS_EventQosPolicy** (p. 739), **DDS_DatabaseQosPolicy** (p. 495), and **DDS_AsynchronousPublisherQosPolicy** (p. 466)). However, the thread settings in the **DDS_ReceiverPoolQosPolicy** (p. 862) control every receive thread created. In this case, there are several schemes to map M threads to N processors; the rotation kind controls which scheme is used.

If **DDS_ThreadSettings_t::cpu_list** (p. 951) is empty, the rotation is irrelevant since no affinity adjustment will occur. Suppose instead that **DDS_ThreadSettings_t::cpu_list** (p. 951) = {0, 1} and that the middleware creates three receive threads: {A, B, C}. If **DDS_ThreadSettings_t::cpu_rotation** (p. 951) is **DDS_THREAD_SETTINGS_CPU_NO_ROTATION** (p. 330), threads A, B and C will have the same processor affinities (0-1), and the OS will control thread scheduling within this bound. It is common to denote CPU affinities as a bitmask, where set bits represent allowed processors to run on. This mask is printed in hex, so a CPU core affinity of 0-1 can be represented by the mask 0x3.

If **DDS_ThreadSettings_t::cpu_rotation** (p. 951) is **DDS_THREAD_SETTINGS_CPU_RR_ROTATION** (p. 330), each thread will be assigned in round-robin fashion to one of the processors in **DDS_ThreadSettings_t::cpu_list** (p. 951); perhaps thread A to 0, B to 1, and C to 0. Note that the order in which internal middleware threads spawn is unspecified.

Not all of these options may be relevant for all operating systems.

Enumerator:

DDS_THREAD_SETTINGS_CPU_NO_ROTATION Any thread controlled by this QoS can run on any listed processor, as determined by OS scheduling.

DDS_THREAD_SETTINGS_CPU_RR_ROTATION Threads controlled by this QoS will be assigned one processor from the list in round-robin order.

5.84 QoS Policies

Quality of Service (QoS) policies.

Modules

- ^ **USER_DATA**

  Attaches a buffer of opaque data that is distributed by means of ***Built-in Topics*** *(p. 42) during discovery.*

- ^ **TOPIC_DATA**

  Attaches a buffer of opaque data that is distributed by means of ***Built-in Topics*** *(p. 42) during discovery.*

- ^ **GROUP_DATA**

  Attaches a buffer of opaque data that is distributed by means of ***Built-in Topics*** *(p. 42) during discovery.*

- ^ **DURABILITY**

  This QoS policy specifies whether or not RTI Connext will store and deliver previously published data samples to new ***DDSDataReader*** *(p. 1087) entities that join the network later.*

- ^ **PRESENTATION**

  Specifies how the samples representing changes to data instances are presented to a subscribing application.

- ^ **DEADLINE**

  Expresses the maximum duration (deadline) within which an instance is expected to be updated.

- ^ **LATENCY_BUDGET**

  Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

- ^ **OWNERSHIP**

  Specifies whether it is allowed for multiple ***DDSDataWriter*** *(p. 1113) (s) to write the same instance of the data and if so, how these modifications should be arbitrated.*

- ^ **OWNERSHIP_STRENGTH**

  Specifies the value of the strength used to arbitrate among multiple ***DDSDataWriter*** *(p. 1113) objects that attempt to modify the same instance of a data type (identified by* ***DDSTopic*** *(p. 1419) + key).*

- **LIVELINESS**

  Specifies and configures the mechanism that allows ***DDSDataReader*** *(p. 1087) entities to detect when* ***DDSDataWriter*** *(p. 1113) entities become disconnected or "dead.".*

- **TIME_BASED_FILTER**

  Filter that allows a ***DDSDataReader*** *(p. 1087) to specify that it is interested only in (potentially) a subset of the values of the data.*

- **PARTITION**

  Set of strings that introduces a logical partition among the topics visible by a ***DDSPublisher*** *(p. 1346) and a* ***DDSSubscriber*** *(p. 1390).*

- **RELIABILITY**

  Indicates the level of reliability offered/requested by RTI Connext.

- **DESTINATION_ORDER**

  Controls the criteria used to determine the logical order among changes made by ***DDSPublisher*** *(p. 1346) entities to the same instance of data (i.e., matching* ***DDSTopic*** *(p. 1419) and key).*

- **HISTORY**

  Specifies the behavior of RTI Connext in the case where the value of an instance changes (one or more times) before it can be successfully communicated to one or more existing subscribers.

- **DURABILITY_SERVICE**

  Various settings to configure the external RTI Persistence Service *used by RTI Connext for DataWriters with a* ***DDS_DurabilityQosPolicy*** *(p. 614) setting of* ***DDS_PERSISTENT_DURABILITY_QOS*** *(p. 349) or* ***DDS_TRANSIENT_DURABILITY_QOS*** *(p. 349).*

- **RESOURCE_LIMITS**

  Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics.

- **TRANSPORT_PRIORITY**

  This QoS policy allows the application to take advantage of transports that are capable of sending messages with different priorities.

- **LIFESPAN**

  Specifies how long the data written by the ***DDSDataWriter*** *(p. 1113) is considered valid.*

- **WRITER_DATA_LIFECYCLE**

  Controls how a DataWriter handles the lifecycle of the instances (keys) that it is registered to manage.

- **READER_DATA_LIFECYCLE**

  Controls how a DataReader manages the lifecycle of the data that it has received.

- **ENTITY_FACTORY**

  A QoS policy for all ***DDSEntity*** *(p. 1253) types that can act as factories for one or more other* ***DDSEntity*** *(p. 1253) types.*

- **Extended Qos Support**

  <<**eXtension**>> (p. 199) *Types and defines used in extended QoS policies.*

- **TRANSPORT_SELECTION**

  <<**eXtension**>> (p. 199) *Specifies the physical transports that a* ***DDSDataWriter*** *(p. 1113) or* ***DDSDataReader*** *(p. 1087) may use to send or receive data.*

- **TRANSPORT_UNICAST**

  <<**eXtension**>> (p. 199) *Specifies a subset of transports and a port number that can be used by an Entity to receive data.*

- **TRANSPORT_MULTICAST**

  <<**eXtension**>> (p. 199) *Specifies the multicast address on which a* ***DDSDataReader*** *(p. 1087) wants to receive its data. It can also specify a port number, as well as a subset of the available (at the* ***DDSDomainParticipant*** *(p. 1139) level) transports with which to receive the multicast data.*

- **TRANSPORT_MULTICAST_MAPPING**

  <<**eXtension**>> (p. 199) *Specifies a list of topic expressions and addresses that can be used by an Entity with a specific topic name to receive data.*

- **DISCOVERY**

  <<**eXtension**>> (p. 199) *Specifies the attributes required to discover participants in the domain.*

- **TRANSPORT_BUILTIN**

  <<**eXtension**>> (p. 199) *Specifies which built-in transports are used.*

- **WIRE_PROTOCOL**

<<**eXtension**>> (p. 199) *Specifies the wire protocol related attributes for the* ***DDSDomainParticipant*** *(p. 1139).*

^ **DATA_READER_RESOURCE_LIMITS**

<<**eXtension**>> (p. 199) *Various settings that configure how DataReaders allocate and use physical memory for internal resources.*

^ **DATA_WRITER_RESOURCE_LIMITS**

<<**eXtension**>> (p. 199) *Various settings that configure how a* ***DDSDataWriter*** *(p. 1113) allocates and uses physical memory for internal resources.*

^ **DATA_READER_PROTOCOL**

<<**eXtension**>> (p. 199) *Specifies the DataReader-specific protocol QoS.*

^ **DATA_WRITER_PROTOCOL**

<<**eXtension**>> (p. 199) *Along with* ***DDS_WireProtocolQosPolicy*** *(p. 1059) and* ***DDS_DataReaderProtocolQosPolicy*** *(p. 501), this QoS policy configures the DDS on-the-network protocol (RTPS).*

^ **SYSTEM_RESOURCE_LIMITS**

<<**eXtension**>> (p. 199) *Configures DomainParticipant-independent resources used by RTI Connext.*

^ **DOMAIN_PARTICIPANT_RESOURCE_LIMITS**

<<**eXtension**>> (p. 199) *Various settings that configure how a* ***DDSDomainParticipant*** *(p. 1139) allocates and uses physical memory for internal resources, including the maximum sizes of various properties.*

^ **EVENT**

<<**eXtension**>> (p. 199) *Configures the internal thread in a DomainParticipant that handles timed events.*

^ **DATABASE**

<<**eXtension**>> (p. 199) *Various threads and resource limits settings used by RTI Connext to control its internal database.*

^ **RECEIVER_POOL**

<<**eXtension**>> (p. 199) *Configures threads used by RTI Connext to receive and process data from transports (for example, UDP sockets).*

^ **PUBLISH_MODE**

<<**eXtension**>> (p. 199) *Specifies how RTI Connext sends application data on the network. This QoS policy can be used to tell RTI Connext to use its* own *thread to send data, instead of the user thread.*

- **DISCOVERY_CONFIG**

  <<**eXtension**>> (p. 199) *Specifies the discovery configuration QoS.*

- **ASYNCHRONOUS_PUBLISHER**

  <<**eXtension**>> (p. 199) *Specifies the asynchronous publishing settings of the* ***DDSPublisher*** *(p. 1346) instances.*

- **TYPESUPPORT**

  <<**eXtension**>> (p. 199) *Allows you to attach application-specific values to a DataWriter or DataReader that are passed to the serialization or deserialization routine of the associated data type.*

- **EXCLUSIVE_AREA**

  <<**eXtension**>> (p. 199) *Configures multi-thread concurrency and deadlock prevention capabilities.*

- **BATCH**

  <<**eXtension**>> (p. 199) *Batch QoS policy used to enable batching in* ***DDSDataWriter*** *(p. 1113) instances.*

- **TYPE_CONSISTENCY_ENFORCEMENT**

  Defines the rules for determining whether the type used to publish a given topic is consistent with that used to subscribe to it.

- **LOCATORFILTER**

  <<**eXtension**>> (p. 199) *The QoS policy used to report the configuration of a MultiChannel DataWriter as part of* ***DDS_PublicationBuiltinTopicData*** *(p. 839).*

- **MULTICHANNEL**

  <<**eXtension**>> (p. 199) *Configures the ability of a DataWriter to send data on different multicast groups (addresses) based on the value of the data.*

- **PROPERTY**

  <<**eXtension**>> (p. 199) *Stores name/value (string) pairs that can be used to configure certain parameters of RTI Connext that are not exposed through formal QoS policies. Can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery.*

- **AVAILABILITY**

  <<**eXtension**>> (p. 199) *Configures the availability of data.*

- **ENTITY_NAME**

<<**eXtension**>> (p. 199) *Assigns a name to a* ***DDSDomainParticipant*** *(p. 1139). This name will be visible during the discovery process and in RTI tools to help you visualize and debug your system.*

- **PROFILE**

  <<**eXtension**>> (p. 199) *Configures the way that XML documents containing QoS profiles are loaded by RTI Connext.*

- **LOGGING**

  <<**eXtension**>> (p. 199) *Configures the RTI Connext logging facility.*

Classes

- struct **DDS_QosPolicyCount**

  Type to hold a counter for a ***DDS_QosPolicyId_t*** *(p. 341).*

- struct **DDS_QosPolicyCountSeq**

  Declares IDL `sequence` < ***DDS_QosPolicyCount*** *(p. 857)* >.

Defines

- #define **DDS_QOS_POLICY_COUNT**

  Number of QoS policies in ***DDS_QosPolicyId_t*** *(p. 341).*

Enumerations

- enum **DDS_QosPolicyId_t** {
  DDS_INVALID_QOS_POLICY_ID,
  DDS_USERDATA_QOS_POLICY_ID,
  DDS_DURABILITY_QOS_POLICY_ID,
  DDS_PRESENTATION_QOS_POLICY_ID,
  DDS_DEADLINE_QOS_POLICY_ID,
  DDS_LATENCYBUDGET_QOS_POLICY_ID,
  DDS_OWNERSHIP_QOS_POLICY_ID,
  DDS_OWNERSHIPSTRENGTH_QOS_POLICY_ID,
  DDS_LIVELINESS_QOS_POLICY_ID,
  DDS_TIMEBASEDFILTER_QOS_POLICY_ID,

DDS_PARTITION_QOS_POLICY_ID,

DDS_RELIABILITY_QOS_POLICY_ID,

DDS_DESTINATIONORDER_QOS_POLICY_ID,

DDS_HISTORY_QOS_POLICY_ID,

DDS_RESOURCELIMITS_QOS_POLICY_ID,

DDS_ENTITYFACTORY_QOS_POLICY_ID,

DDS_WRITERDATALIFECYCLE_QOS_POLICY_ID,

DDS_READERDATALIFECYCLE_QOS_POLICY_ID,

DDS_TOPICDATA_QOS_POLICY_ID,

DDS_GROUPDATA_QOS_POLICY_ID,

DDS_TRANSPORTPRIORITY_QOS_POLICY_ID,

DDS_LIFESPAN_QOS_POLICY_ID,

DDS_DURABILITYSERVICE_QOS_POLICY_ID,

DDS_TYPE_CONSISTENCY_ENFORCEMENT_QOS_-POLICY_ID,

DDS_WIREPROTOCOL_QOS_POLICY_ID,

DDS_DISCOVERY_QOS_POLICY_ID,

DDS_DATAREADERRESOURCELIMITS_QOS_POLICY_ID,

DDS_DATAWRITERRESOURCELIMITS_QOS_POLICY_ID,

DDS_DATAREADERPROTOCOL_QOS_POLICY_ID,

DDS_DATAWRITERPROTOCOL_QOS_POLICY_ID,

DDS_DOMAINPARTICIPANTRESOURCELIMITS_QOS_-POLICY_ID,

DDS_EVENT_QOS_POLICY_ID,

DDS_DATABASE_QOS_POLICY_ID,

DDS_RECEIVERPOOL_QOS_POLICY_ID,

DDS_DISCOVERYCONFIG_QOS_POLICY_ID,

DDS_EXCLUSIVEAREA_QOS_POLICY_ID ,

DDS_SYSTEMRESOURCELIMITS_QOS_POLICY_ID,

DDS_TRANSPORTSELECTION_QOS_POLICY_ID,

DDS_TRANSPORTUNICAST_QOS_POLICY_ID,

DDS_TRANSPORTMULTICAST_QOS_POLICY_ID,

DDS_TRANSPORTBUILTIN_QOS_POLICY_ID,

DDS_TYPESUPPORT_QOS_POLICY_ID,

DDS_PROPERTY_QOS_POLICY_ID,

DDS_PUBLISHMODE_QOS_POLICY_ID,

DDS_ASYNCHRONOUSPUBLISHER_QOS_POLICY_ID,

DDS_ENTITYNAME_QOS_POLICY_ID ,

DDS_BATCH_QOS_POLICY_ID,

DDS_PROFILE_QOS_POLICY_ID,

DDS_LOCATORFILTER_QOS_POLICY_ID,

DDS_MULTICHANNEL_QOS_POLICY_ID ,

DDS_AVAILABILITY_QOS_POLICY_ID,

DDS_TRANSPORTMULTICASTMAPPING_QOS_POLICY_-ID = 1036,

DDS_LOGGING_QOS_POLICY_ID }

Type to identify `QosPolicies`.

5.84.1 Detailed Description

Quality of Service (QoS) policies.

Data Distribution Service (DDS) relies on the use of QoS. A QoS is a set of characteristics that controls some aspect of the behavior of DDS. A QoS is comprised of individual QoS policies (objects conceptually deriving from an *abstract* `QosPolicy` class).

The QosPolicy provides the basic mechanism for an application to specify quality of service parameters. It has an attribute name that is used to uniquely identify each `QosPolicy`.

QosPolicy implementation is comprised of a name, an ID, and a type. The type of a `QosPolicy` value may be atomic, such as an integer or float, or compound (a structure). Compound types are used whenever multiple parameters must be set coherently to define a consistent value for a QosPolicy.

QoS (i.e., a list of `QosPolicy` objects) may be associated with all **DDSEntity** (p. 1253) objects in the system such as **DDSTopic** (p. 1419), **DDSDataWriter** (p. 1113), **DDSDataReader** (p. 1087), **DDSPublisher** (p. 1346), **DDSSubscriber** (p. 1390), and **DDSDomainParticipant** (p. 1139).

5.84.2 Specifying QoS on entities

QosPolicies can be set programmatically when an **DDSEntity** (p. 1253) is created, or modified with the **DDSEntity** (p. 1253)'s **set_qos (abstract)** (p. 1254) method.

QosPolicies can also be configured from XML resources (files, strings). With this approach, you can change the QoS without recompiling the application. For more information, see **Configuring QoS Profiles with XML** (p. 151).

To customize a **DDSEntity** (p. 1253)'s QoS before creating the entity, the correct pattern is:

- First, initialize a QoS object with the appropriate INITIALIZER constructor.
- Call the relevant get_<entity>_default_qos() method.
- Modify the QoS values as desired.
- Finally, create the entity.

Each `QosPolicy` is treated independently from the others. This approach has the advantage of being very extensible. However, there may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified via the **set_qos (abstract)** (p. 1254) operation, or when the **DDSEntity** (p. 1253) is created.

When a policy is changed after being set to a given value, it is not required that the new value be applied instantaneously; RTI Connext is allowed to apply it after a transition phase. In addition, some `QosPolicy` have immutable semantics, meaning that they can only be specified either at **DDSEntity** (p. 1253) creation time or else prior to calling the **DDSEntity::enable** (p. 1256) operation on the entity.

Each **DDSEntity** (p. 1253) can be configured with a list of `QosPolicy` objects. However, not all QosPolicies are supported by each **DDSEntity** (p. 1253). For instance, a **DDSDomainParticipant** (p. 1139) supports a different set of QosPolicies than a **DDSTopic** (p. 1419) or a **DDSPublisher** (p. 1346).

5.84.3 QoS compatibility

In several cases, for communications to occur properly (or efficiently), a QosPolicy on the publisher side must be compatible with a corresponding policy on the subscriber side. For example, if a **DDSSubscriber** (p. 1390) requests to receive data reliably while the corresponding **DDSPublisher** (p. 1346) defines a best-effort policy, communication will not happen as requested.

To address this issue and maintain the desirable decoupling of publication and subscription as much as possible, the QosPolicy specification follows the **subscriber-requested, publisher-offered pattern**.

In this pattern, the subscriber side can specify a "requested" value for a particular `QosPolicy`. The publisher side specifes an "offered" value for that

QosPolicy. RTI Connext will then determine whether the value requested by the subscriber side is compatible with what is offered by the publisher side. If the two policies are compatible, then communication will be established. If the two policies are not compatible, RTI Connext will not establish communications between the two **DDSEntity** (p. 1253) objects and will record this fact by means of the **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323) on the publisher end and **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323) on the subscriber end. The application can detect this fact by means of a **DDSListener** (p. 1318) or a **DDSCondition** (p. 1075).

The following **properties** are defined on a QosPolicy.

- **RxO (p. 340) property**

  The QosPolicy objects that need to be set in a compatible manner between the **publisher** and **subscriber** end are indicated by the setting of the **RxO** (p. 340) property:

  - **RxO** (p. 340) = **YES** indicates that the policy can be set both at the publishing and subscribing ends and the values must be set in a compatible manner. In this case the compatible values are explicitly defined.
  - **RxO** (p. 340) = **NO** indicates that the policy can be set both at the publishing and subscribing ends but the two settings are independent. That is, all combinations of values are compatible.
  - **RxO** (p. 340) = **N/A** indicates that the policy can only be specified at either the publishing or the subscribing end, but not at both ends. So compatibility does not apply.

- **Changeable (p. 340) property**

  Determines whether a QosPolicy can be changed.

  NO (p. 340) – policy can only be specified at **DDSEntity** (p. 1253) creation time.

  UNTIL ENABLE (p. 340) – policy can only be changed before the **DDSEntity** (p. 1253) is enabled.

  YES (p. 340) – policy can be changed at any time.

5.84.4 Define Documentation

5.84.4.1 #define DDS_QOS_POLICY_COUNT

Number of QoS policies in **DDS_QosPolicyId_t** (p. 341).

5.84.5 Enumeration Type Documentation

5.84.5.1 enum DDS_QosPolicyId_t

Type to identify `QosPolicies`.

Enumerator:

DDS_INVALID_QOS_POLICY_ID Identifier for an invalid QoS policy.

DDS_USERDATA_QOS_POLICY_ID Identifier for **DDS_UserDataQosPolicy** (p. 1048).

DDS_DURABILITY_QOS_POLICY_ID Identifier for **DDS_DurabilityQosPolicy** (p. 614).

DDS_PRESENTATION_QOS_POLICY_ID Identifier for **DDS_PresentationQosPolicy** (p. 823).

DDS_DEADLINE_QOS_POLICY_ID Identifier for **DDS_DeadlineQosPolicy** (p. 567).

DDS_LATENCYBUDGET_QOS_POLICY_ID Identifier for **DDS_LatencyBudgetQosPolicy** (p. 771).

DDS_OWNERSHIP_QOS_POLICY_ID Identifier for **DDS_OwnershipQosPolicy** (p. 807).

DDS_OWNERSHIPSTRENGTH_QOS_POLICY_ID Identifier for **DDS_OwnershipStrengthQosPolicy** (p. 814).

DDS_LIVELINESS_QOS_POLICY_ID Identifier for **DDS_LivelinessQosPolicy** (p. 779).

DDS_TIMEBASEDFILTER_QOS_POLICY_ID Identifier for **DDS_TimeBasedFilterQosPolicy** (p. 954).

DDS_PARTITION_QOS_POLICY_ID Identifier for **DDS_PartitionQosPolicy** (p. 820).

DDS_RELIABILITY_QOS_POLICY_ID Identifier for **DDS_ReliabilityQosPolicy** (p. 865).

DDS_DESTINATIONORDER_QOS_POLICY_ID Identifier for **DDS_DestinationOrderQosPolicy** (p. 570).

DDS_HISTORY_QOS_POLICY_ID Identifier for **DDS_HistoryQosPolicy** (p. 758).

DDS_RESOURCELIMITS_QOS_POLICY_ID Identifier for **DDS_ResourceLimitsQosPolicy** (p. 879).

DDS_ENTITYFACTORY_QOS_POLICY_ID Identifier for **DDS_EntityFactoryQosPolicy** (p. 733).

DDS_WRITERDATALIFECYCLE_QOS_POLICY_ID Identifier for **DDS_WriterDataLifecycleQosPolicy** (p. 1071).

DDS_READERDATALIFECYCLE_QOS_POLICY_ID Identifier for **DDS_ReaderDataLifecycleQosPolicy** (p. 859).

DDS_TOPICDATA_QOS_POLICY_ID Identifier for **DDS_TopicDataQosPolicy** (p. 963).

DDS_GROUPDATA_QOS_POLICY_ID Identifier for **DDS_GroupDataQosPolicy** (p. 755).

DDS_TRANSPORTPRIORITY_QOS_POLICY_ID Identifier for **DDS_TransportPriorityQosPolicy** (p. 983).

DDS_LIFESPAN_QOS_POLICY_ID Identifier for **DDS_LifespanQosPolicy** (p. 773).

DDS_DURABILITYSERVICE_QOS_POLICY_ID Identifier for **DDS_DurabilityServiceQosPolicy** (p. 618).

DDS_TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_ID Identifier for **DDS_TypeConsistencyEnforcementQosPolicy** (p. 1038).

DDS_WIREPROTOCOL_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_WireProtocolQosPolicy** (p. 1059)

DDS_DISCOVERY_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DiscoveryQosPolicy** (p. 582)

DDS_DATAREADERRESOURCELIMITS_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DataReaderResourceLimitsQosPolicy** (p. 521)

DDS_DATAWRITERRESOURCELIMITS_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DataWriterResourceLimitsQosPolicy** (p. 560)

DDS_DATAREADERPROTOCOL_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DataReaderProtocolQosPolicy** (p. 501)

DDS_DATAWRITERPROTOCOL_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DataWriterProtocolQosPolicy** (p. 535)

DDS_DOMAINPARTICIPANTRESOURCELIMITS_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DomainParticipantResourceLimitsQosPolicy** (p. 593)

DDS_EVENT_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_EventQosPolicy** (p. 739)

DDS_DATABASE_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DatabaseQosPolicy** (p. 495)

DDS_RECEIVERPOOL_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_ReceiverPoolQosPolicy** (p. 862)

DDS_DISCOVERYCONFIG_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_DiscoveryConfigQosPolicy** (p. 573)

DDS_EXCLUSIVEAREA_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_ExclusiveAreaQosPolicy** (p. 742)

DDS_SYSTEMRESOURCELIMITS_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_SystemResourceLimitsQosPolicy** (p. 948)

DDS_TRANSPORTSELECTION_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_TransportSelectionQosPolicy** (p. 985)

DDS_TRANSPORTUNICAST_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_TransportUnicastQosPolicy** (p. 987)

DDS_TRANSPORTMULTICAST_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_TransportMulticastQosPolicy** (p. 978)

DDS_TRANSPORTBUILTIN_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_TransportBuiltinQosPolicy** (p. 969)

DDS_TYPESUPPORT_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_TypeSupportQosPolicy** (p. 1040)

DDS_PROPERTY_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_PropertyQosPolicy** (p. 834)

DDS_PUBLISHMODE_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_PublishModeQosPolicy** (p. 853)

DDS_ASYNCHRONOUSPUBLISHER_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_AsynchronousPublisherQosPolicy** (p. 466)

DDS_ENTITYNAME_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_EntityNameQosPolicy** (p. 735)

DDS_BATCH_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_BatchQosPolicy** (p. 476)

DDS_PROFILE_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_ProfileQosPolicy** (p. 830)

DDS_LOCATORFILTER_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_LocatorFilterQosPolicy** (p. 787)

DDS_MULTICHANNEL_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_MultiChannelQosPolicy** (p. 796)

DDS_AVAILABILITY_QOS_POLICY_ID *<<eXtension>>* *(p. 199)* Identifier for **DDS_AvailabilityQosPolicy** (p. 471)

DDS_TRANSPORTMULTICASTMAPPING_QOS_POLICY_ID

DDS_LOGGING_QOS_POLICY_ID *<<eXtension>>* *(p. 199)*
Identifier for **DDS_LoggingQosPolicy** (p. 791)

5.85 USER_DATA

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Classes

- struct **DDS_UserDataQosPolicy**

  *Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.*

Variables

- const char *const **DDS_USERDATA_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_UserDataQosPolicy** (p. 1048).*

5.85.1 Detailed Description

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

5.85.2 Variable Documentation

5.85.2.1 const char* const DDS_USERDATA_QOS_POLICY_NAME

Stringified human-readable name for **DDS_UserDataQosPolicy** (p. 1048).

5.86 TOPIC_DATA

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Classes

- struct **DDS_TopicDataQosPolicy**

  *Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.*

Variables

- const char *const **DDS_TOPICDATA_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_TopicDataQosPolicy** (p. 963).*

5.86.1 Detailed Description

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

5.86.2 Variable Documentation

5.86.2.1 const char* const DDS_TOPICDATA_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TopicDataQosPolicy** (p. 963).

5.87 GROUP_DATA

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Classes

- struct **DDS_GroupDataQosPolicy**

  *Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.*

Variables

- const char *const **DDS_GROUPDATA_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_GroupDataQosPolicy** (p. 755).*

5.87.1 Detailed Description

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

5.87.2 Variable Documentation

5.87.2.1 const char* const DDS_GROUPDATA_QOS_POLICY_NAME

Stringified human-readable name for **DDS_GroupDataQosPolicy** (p. 755).

5.88 DURABILITY

This QoS policy specifies whether or not RTI Connext will store and deliver previously published data samples to new **DDSDataReader** (p. 1087) entities that join the network later.

Classes

- struct **DDS_DurabilityQosPolicy**

  *This QoS policy specifies whether or not RTI Connext will store and deliver previously published data samples to new **DDSDataReader** (p. 1087) entities that join the network later.*

Enumerations

- enum **DDS_DurabilityQosPolicyKind** {
  DDS_VOLATILE_DURABILITY_QOS,
  DDS_TRANSIENT_LOCAL_DURABILITY_QOS,
  DDS_TRANSIENT_DURABILITY_QOS,
  DDS_PERSISTENT_DURABILITY_QOS }

  Kinds of durability.

Variables

- const char *const **DDS_DURABILITY_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_DurabilityQosPolicy** (p. 614).*

5.88.1 Detailed Description

This QoS policy specifies whether or not RTI Connext will store and deliver previously published data samples to new **DDSDataReader** (p. 1087) entities that join the network later.

5.88.2 Enumeration Type Documentation

5.88.2.1 enum DDS_DurabilityQosPolicyKind

Kinds of durability.

QoS:

DDS_DurabilityQosPolicy (p. 614)

Enumerator:

DDS_VOLATILE_DURABILITY_QOS **[default]** RTI Connext does not need to keep any samples of data instances on behalf of any **DDSDataReader** (p. 1087) that is unknown by the **DDSDataWriter** (p. 1113) at the time the instance is written.

In other words, RTI Connext will only attempt to provide the data to existing subscribers.

This option does not require RTI Persistence Service.

DDS_TRANSIENT_LOCAL_DURABILITY_QOS RTI Connext will attempt to keep some samples so that they can be delivered to any potential late-joining **DDSDataReader** (p. 1087).

Which particular samples are kept depends on other QoS such as **DDS_HistoryQosPolicy** (p. 758) and **DDS_ResourceLimitsQosPolicy** (p. 879). RTI Connext is only required to keep the data in memory of the **DDSDataWriter** (p. 1113) that wrote the data.

Data is not required to survive the **DDSDataWriter** (p. 1113).

For this setting to be effective, you must also set the **DDS_ReliabilityQosPolicy::kind** (p. 868) to **DDS_RELIABLE_RELIABILITY_QOS** (p. 363).

This option does not require RTI Persistence Service.

DDS_TRANSIENT_DURABILITY_QOS RTI Connext will attempt to keep some samples so that they can be delivered to any potential late-joining **DDSDataReader** (p. 1087).

Which particular samples are kept depends on other QoS such as **DDS_HistoryQosPolicy** (p. 758) and **DDS_ResourceLimitsQosPolicy** (p. 879). RTI Connext is only required to keep the data in memory and not in permanent storage.

Data is not tied to the lifecycle of the **DDSDataWriter** (p. 1113).

Data will survive the **DDSDataWriter** (p. 1113).

This option requires RTI Persistence Service.

DDS_PERSISTENT_DURABILITY_QOS Data is kept on permanent storage, so that they can outlive a system session.

This option requires RTI Persistence Service.

5.88.3 Variable Documentation

5.88.3.1 const char* const DDS_DURABILITY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DurabilityQosPolicy** (p. 614).

5.89 PRESENTATION

Specifies how the samples representing changes to data instances are presented to a subscribing application.

Classes

- struct **DDS_PresentationQosPolicy**

  Specifies how the samples representing changes to data instances are presented to a subscribing application.

Enumerations

- enum **DDS_PresentationQosPolicyAccessScopeKind** {
  DDS_INSTANCE_PRESENTATION_QOS,
  DDS_TOPIC_PRESENTATION_QOS,
  DDS_GROUP_PRESENTATION_QOS,
  DDS_HIGHEST_OFFERED_PRESENTATION_QOS }

  Kinds of presentation "access scope".

Variables

- const char *const **DDS_PRESENTATION_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_PresentationQosPolicy*** (p. 823).

5.89.1 Detailed Description

Specifies how the samples representing changes to data instances are presented to a subscribing application.

5.89.2 Enumeration Type Documentation

5.89.2.1 enum DDS_PresentationQosPolicyAccessScopeKind

Kinds of presentation "access scope".

Access scope determines the largest scope spanning the entities for which the order and coherency of changes can be preserved.

QoS:

DDS_PresentationQosPolicy (p. 823)

Enumerator:

DDS_INSTANCE_PRESENTATION_QOS [**default**] Scope spans only a single instance.

Indicates that changes to one instance need not be coherent nor ordered with respect to changes to any other instance. In other words, order and coherent changes apply to each instance separately.

DDS_TOPIC_PRESENTATION_QOS Scope spans to all instances within the same **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)), but not across instances in different **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)).

DDS_GROUP_PRESENTATION_QOS Scope spans to all instances belonging to **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)) entities within the same **DDSPublisher** (p. 1346) (or **DDSSubscriber** (p. 1390)).

DDS_HIGHEST_OFFERED_PRESENTATION_QOS This value only applies to a **DDSSubscriber** (p. 1390). The **DDSSubscriber** (p. 1390) will use the access scope specified by each remote **DDSPublisher** (p. 1346).

5.89.3 Variable Documentation

5.89.3.1 const char* const DDS_PRESENTATION_QOS_POLICY_NAME

Stringified human-readable name for **DDS_PresentationQosPolicy** (p. 823).

5.90 DEADLINE

Expresses the maximum duration (deadline) within which an instance is expected to be updated.

Classes

- struct **DDS_DeadlineQosPolicy**

  Expresses the maximum duration (deadline) within which an instance is expected to be updated.

Variables

- const char *const **DDS_DEADLINE_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_DeadlineQosPolicy*** *(p. 567).*

5.90.1 Detailed Description

Expresses the maximum duration (deadline) within which an instance is expected to be updated.

5.90.2 Variable Documentation

5.90.2.1 const char* const DDS_DEADLINE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DeadlineQosPolicy** (p. 567).

5.91 LATENCY_BUDGET

Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

Classes

- struct **DDS_LatencyBudgetQosPolicy**

  Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

Variables

- const char *const **DDS_LATENCYBUDGET_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_LatencyBudgetQosPolicy*** (p. 771).

5.91.1 Detailed Description

Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

5.91.2 Variable Documentation

5.91.2.1 const char* const DDS_LATENCYBUDGET_QOS_POLICY_NAME

Stringified human-readable name for **DDS_LatencyBudgetQosPolicy** (p. 771).

5.92 OWNERSHIP

Specifies whether it is allowed for multiple **DDSDataWriter** (p. 1113) (s) to write the same instance of the data and if so, how these modifications should be arbitrated.

Classes

- struct **DDS_OwnershipQosPolicy**

  *Specifies whether it is allowed for multiple **DDSDataWriter** (p. 1113) (s) to write the same instance of the data and if so, how these modifications should be arbitrated.*

Enumerations

- enum **DDS_OwnershipQosPolicyKind** {
  DDS_SHARED_OWNERSHIP_QOS,
  DDS_EXCLUSIVE_OWNERSHIP_QOS }

  Kinds of ownership.

Variables

- const char *const **DDS_OWNERSHIP_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_OwnershipQosPolicy** (p. 807).*

5.92.1 Detailed Description

Specifies whether it is allowed for multiple **DDSDataWriter** (p. 1113) (s) to write the same instance of the data and if so, how these modifications should be arbitrated.

5.92.2 Enumeration Type Documentation

5.92.2.1 enum DDS_OwnershipQosPolicyKind

Kinds of ownership.

QoS:

DDS_OwnershipQosPolicy (p. 807)

Enumerator:

DDS_SHARED_OWNERSHIP_QOS **[default]** Indicates shared ownership for each instance.

Multiple writers are allowed to update the same instance and all the updates are made available to the readers. In other words there is no concept of an owner for the instances.

This is the **default** behavior if the **OWNERSHIP** (p. 355) policy is not specified or supported.

DDS_EXCLUSIVE_OWNERSHIP_QOS Indicates each instance can only be owned by one **DDSDataWriter** (p. 1113), but the owner of an instance can change dynamically.

The selection of the owner is controlled by the setting of the **OWNERSHIP_STRENGTH** (p. 357) policy. The owner is always set to be the highest-strength **DDSDataWriter** (p. 1113) object among the ones currently active (as determined by the **LIVELINESS** (p. 358)).

5.92.3 Variable Documentation

5.92.3.1 const char* const DDS_OWNERSHIP_QOS_POLICY_NAME

Stringified human-readable name for **DDS_OwnershipQosPolicy** (p. 807).

5.93 OWNERSHIP_STRENGTH

Specifies the value of the strength used to arbitrate among multiple **DDSDataWriter** (p. 1113) objects that attempt to modify the same instance of a data type (identified by **DDSTopic** (p. 1419) + key).

Classes

- struct **DDS_OwnershipStrengthQosPolicy**

  *Specifies the value of the strength used to arbitrate among multiple **DDSDataWriter** (p. 1113) objects that attempt to modify the same instance of a data type (identified by **DDSTopic** (p. 1419) + key).*

Variables

- const char *const **DDS_OWNERSHIPSTRENGTH_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_OwnershipStrengthQosPolicy** (p. 814).*

5.93.1 Detailed Description

Specifies the value of the strength used to arbitrate among multiple **DDSDataWriter** (p. 1113) objects that attempt to modify the same instance of a data type (identified by **DDSTopic** (p. 1419) + key).

5.93.2 Variable Documentation

5.93.2.1 const char* const DDS_OWNERSHIPSTRENGTH_QOS_POLICY_NAME

Stringified human-readable name for **DDS_OwnershipStrengthQosPolicy** (p. 814).

5.94 LIVELINESS

Specifies and configures the mechanism that allows **DDSDataReader** (p. 1087) entities to detect when **DDSDataWriter** (p. 1113) entities become disconnected or "dead.".

Classes

- struct **DDS_LivelinessQosPolicy**

  *Specifies and configures the mechanism that allows **DDSDataReader** (p. 1087) entities to detect when **DDSDataWriter** (p. 1113) entities become disconnected or "dead.".*

Enumerations

- enum **DDS_LivelinessQosPolicyKind** {
  DDS_AUTOMATIC_LIVELINESS_QOS,
  DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
  DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS }

  Kinds of liveliness.

Variables

- const char *const **DDS_LIVELINESS_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_LivelinessQosPolicy** (p. 779).*

5.94.1 Detailed Description

Specifies and configures the mechanism that allows **DDSDataReader** (p. 1087) entities to detect when **DDSDataWriter** (p. 1113) entities become disconnected or "dead.".

5.94.2 Enumeration Type Documentation

5.94.2.1 enum DDS_LivelinessQosPolicyKind

Kinds of liveliness.

QoS:

DDS_LivelinessQosPolicy (p. 779)

Enumerator:

DDS_AUTOMATIC_LIVELINESS_QOS [**default**] The infrastructure will automatically signal liveliness for the **DDSDataWriter** (p. 1113) (s) at least as often as required by the `lease_duration`.

A **DDSDataWriter** (p. 1113) with this setting does not need to take any specific action in order to be considered 'alive.' The **DDSDataWriter** (p. 1113) is only 'not alive' when the participant to which it belongs terminates (gracefully or not), or when there is a network problem that prevents the current participant from contacting that remote participant.

DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS RTI Connext will assume that as long as at least one **DDSDataWriter** (p. 1113) belonging to the **DDSDomainParticipant** (p. 1139) (or the **DDSDomainParticipant** (p. 1139) itself) has asserted its liveliness, then the other Entities belonging to that same **DDSDomainParticipant** (p. 1139) are also alive.

The user application takes responsibility to signal liveliness to RTI Connext either by calling **DDSDomainParticipant::assert_liveliness** (p. 1192), or by calling **DDSDataWriter::assert_liveliness** (p. 1121), or **FooDataWriter::write** (p. 1484) on any **DDSDataWriter** (p. 1113) belonging to the **DDSDomainParticipant** (p. 1139).

DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS RTI Connext will only assume liveliness of the **DDSDataWriter** (p. 1113) if the application has asserted liveliness of that **DDSDataWriter** (p. 1113) itself.

The user application takes responsibility to signal liveliness to RTI Connext using the **DDSDataWriter::assert_liveliness** (p. 1121) method, or by writing some data.

5.94.3 Variable Documentation

5.94.3.1 const char* const DDS_LIVELINESS_QOS_POLICY_NAME

Stringified human-readable name for **DDS_LivelinessQosPolicy** (p. 779).

5.95 TIME_BASED_FILTER

Filter that allows a **DDSDataReader** (p. 1087) to specify that it is interested only in (potentially) a subset of the values of the data.

Classes

- struct **DDS_TimeBasedFilterQosPolicy**

  *Filter that allows a **DDSDataReader** (p. 1087) to specify that it is interested only in (potentially) a subset of the values of the data.*

Variables

- const char *const **DDS_TIMEBASEDFILTER_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_TimeBasedFilterQosPolicy** (p. 954).*

5.95.1 Detailed Description

Filter that allows a **DDSDataReader** (p. 1087) to specify that it is interested only in (potentially) a subset of the values of the data.

5.95.2 Variable Documentation

5.95.2.1 const char* const DDS_TIMEBASEDFILTER_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TimeBasedFilterQosPolicy** (p. 954).

5.96 PARTITION

Set of strings that introduces a logical partition among the topics visible by a **DDSPublisher** (p. 1346) and a **DDSSubscriber** (p. 1390).

Classes

- struct **DDS_PartitionQosPolicy**

  *Set of strings that introduces a logical partition among the topics visible by a **DDSPublisher** (p. 1346) and a **DDSSubscriber** (p. 1390).*

Variables

- const char *const **DDS_PARTITION_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_PartitionQosPolicy** (p. 820).*

5.96.1 Detailed Description

Set of strings that introduces a logical partition among the topics visible by a **DDSPublisher** (p. 1346) and a **DDSSubscriber** (p. 1390).

5.96.2 Variable Documentation

5.96.2.1 const char* const DDS_PARTITION_QOS_POLICY_NAME

Stringified human-readable name for **DDS_PartitionQosPolicy** (p. 820).

5.97 RELIABILITY

Indicates the level of reliability offered/requested by RTI Connext.

Classes

- struct **DDS_ReliabilityQosPolicy**
  Indicates the level of reliability offered/requested by RTI Connext.

Enumerations

- enum **DDS_ReliabilityQosPolicyKind** {
  DDS_BEST_EFFORT_RELIABILITY_QOS,
  DDS_RELIABLE_RELIABILITY_QOS }
  Kinds of reliability.

- enum **DDS_ReliabilityQosPolicyAcknowledgmentModeKind** {
  DDS_PROTOCOL_ACKNOWLEDGMENT_MODE,
  DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE ,
  DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_-MODE }
  Kinds of acknowledgment.

Variables

- const char *const **DDS_RELIABILITY_QOS_POLICY_NAME**
  *Stringified human-readable name for **DDS_ReliabilityQosPolicy*** (p. 865).

5.97.1 Detailed Description

Indicates the level of reliability offered/requested by RTI Connext.

5.97.2 Enumeration Type Documentation

5.97.2.1 enum DDS_ReliabilityQosPolicyKind

Kinds of reliability.

QoS:

DDS_ReliabilityQosPolicy (p. 865)

Enumerator:

DDS_BEST_EFFORT_RELIABILITY_QOS Indicates that it is acceptable to not retry propagation of any samples.

Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples.

[**default**] for **DDSDataReader** (p. 1087) and **DDSTopic** (p. 1419)

DDS_RELIABLE_RELIABILITY_QOS Specifies RTI Connext will attempt to deliver all samples in its history. Missed samples may be retried.

In steady-state (no modifications communicated via the **DDSDataWriter** (p. 1113)), RTI Connext guarantees that all samples in the **DDSDataWriter** (p. 1113) history will eventually be delivered to all the **DDSDataReader** (p. 1087) objects (subject to timeouts that indicate loss of communication with a particular **DDSSubscriber** (p. 1390)).

Outside steady state, the **HISTORY** (p. 367) and **RESOURCE_LIMITS** (p. 371) policies will determine how samples become part of the history and whether samples can be discarded from it.

[**default**] for **DDSDataWriter** (p. 1113)

5.97.2.2 enum DDS_ReliabilityQosPolicyAcknowledgmentModeKind

Kinds of acknowledgment.

QoS:

DDS_ReliabilityQosPolicy (p. 865)

Enumerator:

DDS_PROTOCOL_ACKNOWLEDGMENT_MODE Samples are acknowledged by RTPS protocol.

Samples are acknowledged according to the Real-Time Publish-Subscribe (RTPS) interoperability protocol.

DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE Samples are acknowleged automatically after a subscribing application has accessed them.

A sample received by a **FooDataReader** (p. 1444) is acknowledged after it has been taken and then returned. Specifically,

all samples taken by a call to **FooDataReader::take** (p. 1448) are acknowledged after **FooDataReader::return_loan** (p. 1471) is called. Acknowledgments are sent at a rate determinded by **DDS_RtpsReliableReaderProtocol_t::samples_per_app_ack** (p. 887).

DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE
Samples are acknowledged after the subscribing application explicitly calls acknowledge on the samples.

Samples received by a **DDSDataReader** (p. 1087) are explicitly acknowledged by the subscribing application, after it calls either **DDSDataReader::acknowledge_all** (p. 1095) or **DDSDataReader::acknowledge_sample** (p. 1095).

5.97.3 Variable Documentation

5.97.3.1 const char* const DDS_RELIABILITY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_ReliabilityQosPolicy** (p. 865).

5.98 DESTINATION_ORDER

Controls the criteria used to determine the logical order among changes made by **DDSPublisher** (p. 1346) entities to the same instance of data (i.e., matching **DDSTopic** (p. 1419) and key).

Classes

- struct **DDS_DestinationOrderQosPolicy**

  *Controls how the middleware will deal with data sent by multiple **DDSDataWriter** (p. 1113) entities for the same instance of data (i.e., same **DDSTopic** (p. 1419) and key).*

Enumerations

- enum **DDS_DestinationOrderQosPolicyKind** {

  DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,

  DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS }

  Kinds of destination order.

Variables

- const char *const **DDS_DESTINATIONORDER_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_DestinationOrderQosPolicy** (p. 570).*

5.98.1 Detailed Description

Controls the criteria used to determine the logical order among changes made by **DDSPublisher** (p. 1346) entities to the same instance of data (i.e., matching **DDSTopic** (p. 1419) and key).

5.98.2 Enumeration Type Documentation

5.98.2.1 enum DDS_DestinationOrderQosPolicyKind

Kinds of destination order.

QoS:

DDS_DestinationOrderQosPolicy (p. 570)

Enumerator:

DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
[**default**] Indicates that data is ordered based on the reception time at each **DDSSubscriber** (p. 1390).

Since each subscriber may receive the data at different times there is no guaranteed that the changes will be seen in the same order. Consequently, it is possible for each subscriber to end up with a different final value for the data.

DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
Indicates that data is ordered based on a time-stamp placed at the source (by RTI Connext or by the application).

In any case this guarantees a consistent final value for the data in all subscribers.

See also:

Special Instructions if Using 'Timestamp' APIs and BY_SOURCE_TIMESTAMP Destination Ordering: (p. 1358)

5.98.3 Variable Documentation

5.98.3.1 const char* const DDS_DESTINATIONORDER_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DestinationOrderQosPolicy** (p. 570).

5.99 HISTORY

Specifies the behavior of RTI Connext in the case where the value of an instance changes (one or more times) before it can be successfully communicated to one or more existing subscribers.

Classes

- struct **DDS_HistoryQosPolicy**

  Specifies the behavior of RTI Connext in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers.

Enumerations

- enum **DDS_HistoryQosPolicyKind** {
  DDS_KEEP_LAST_HISTORY_QOS,
  DDS_KEEP_ALL_HISTORY_QOS }

  Kinds of history.

- enum **DDS_RefilterQosPolicyKind** {
  DDS_NONE_REFILTER_QOS,
  DDS_ALL_REFILTER_QOS,
  DDS_ON_DEMAND_REFILTER_QOS }

  <<**eXtension**>> (p. 199) *Kinds of Refiltering*

Variables

- const char *const **DDS_HISTORY_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_HistoryQosPolicy*** *(p. 758).*

5.99.1 Detailed Description

Specifies the behavior of RTI Connext in the case where the value of an instance changes (one or more times) before it can be successfully communicated to one or more existing subscribers.

5.99.2 Enumeration Type Documentation

5.99.2.1 enum DDS_HistoryQosPolicyKind

Kinds of history.

QoS:

DDS_HistoryQosPolicy (p. 758)

Enumerator:

DDS_KEEP_LAST_HISTORY_QOS **[default]** Keep the last `depth` samples.

On the publishing side, RTI Connext will only attempt to keep the most recent `depth` samples of each instance of data (identified by its key) managed by the **DDSDataWriter** (p. 1113).

On the subscribing side, the **DDSDataReader** (p. 1087) will only attempt to keep the most recent `depth` samples received for each instance (identified by its key) until the application takes them via the **DDSDataReader** (p. 1087) 's **take()** operation.

DDS_KEEP_ALL_HISTORY_QOS Keep *all* the samples.

On the publishing side, RTI Connext will attempt to keep all samples (representing each value written) of each instance of data (identified by its key) managed by the **DDSDataWriter** (p. 1113) until they can be delivered to all subscribers.

On the subscribing side, RTI Connext will attempt to keep all samples of each instance of data (identified by its key) managed by the **DDSDataReader** (p. 1087). These samples are kept until the application takes them from RTI Connext via the **take()** operation.

5.99.2.2 enum DDS_RefilterQosPolicyKind

*<<**eXtension**>> (p. 199)* Kinds of Refiltering

QoS:

DDS_HistoryQosPolicy (p. 758)

Enumerator:

DDS_NONE_REFILTER_QOS **[default]** Do not filter existing samples for a new reader

On the publishing side, when a new reader matches a writer, the writer can be configured to filter previously written samples stored in the

writer queue for the new reader. This option configures the writer to not filter any existing samples for the reader and the reader will do the filtering.

DDS_ALL_REFILTER_QOS Filter all existing samples for a new reader.

On the publishing side, when a new reader matches a writer, the writer can be configured to filter previously written samples stored in the writer queue. This option configures the writer to filter all existing samples for the reader when a new reader is matched to the writer.

DDS_ON_DEMAND_REFILTER_QOS Filter existing samples only when they are requested by the reader.

On the publishing side, when a new reader matches a writer, the writer can be configured to filter previously written samples stored in the writer queue. This option configures the writer to filter only existing samples that are requested by the reader.

5.99.3 Variable Documentation

5.99.3.1 const char* const DDS_HISTORY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_HistoryQosPolicy** (p. 758).

5.100 DURABILITY_SERVICE

Various settings to configure the external *RTI Persistence Service* used by RTI Connext for DataWriters with a **DDS_DurabilityQosPolicy** (p. 614) setting of **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) or **DDS_TRANSIENT_DURABILITY_QOS** (p. 349).

Classes

- struct **DDS_DurabilityServiceQosPolicy**

  Various settings to configure the external RTI Persistence Service *used by RTI Connext for DataWriters with a* ***DDS_DurabilityQosPolicy*** *(p. 614) setting of* ***DDS_PERSISTENT_DURABILITY_QOS*** *(p. 349) or* ***DDS_TRANSIENT_DURABILITY_QOS*** *(p. 349).*

Variables

- const char *const **DDS_DURABILITYSERVICE_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_DurabilityServiceQosPolicy*** *(p. 618).*

5.100.1 Detailed Description

Various settings to configure the external *RTI Persistence Service* used by RTI Connext for DataWriters with a **DDS_DurabilityQosPolicy** (p. 614) setting of **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) or **DDS_TRANSIENT_DURABILITY_QOS** (p. 349).

5.100.2 Variable Documentation

5.100.2.1 const char* const DDS_DURABILITYSERVICE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DurabilityServiceQosPolicy** (p. 618).

5.101 RESOURCE_LIMITS

Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics.

Classes

- struct **DDS_ResourceLimitsQosPolicy**

  Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics.

Variables

- const char *const **DDS_RESOURCELIMITS_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_ResourceLimitsQosPolicy*** *(p. 879).*

- const **DDS_Long DDS_LENGTH_UNLIMITED**

  A special value indicating an unlimited quantity.

5.101.1 Detailed Description

Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics.

5.101.2 Variable Documentation

5.101.2.1 const char* const DDS_RESOURCELIMITS_QOS_POLICY_NAME

Stringified human-readable name for **DDS_ResourceLimitsQosPolicy** (p. 879).

5.101.2.2 const DDS_Long DDS_LENGTH_UNLIMITED

A special value indicating an unlimited quantity.

Examples:

HelloWorld_subscriber.cxx.

5.102 TRANSPORT_PRIORITY

This QoS policy allows the application to take advantage of transports that are capable of sending messages with different priorities.

Classes

- struct **DDS_TransportPriorityQosPolicy**

  This QoS policy allows the application to take advantage of transports that are capable of sending messages with different priorities.

Variables

- const char *const **DDS_TRANSPORTPRIORITY_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_TransportPriorityQosPolicy** (p. 983).*

5.102.1 Detailed Description

This QoS policy allows the application to take advantage of transports that are capable of sending messages with different priorities.

5.102.2 Variable Documentation

5.102.2.1 const char* const DDS_TRANSPORTPRIORITY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TransportPriorityQosPolicy** (p. 983).

5.103 LIFESPAN

Specifies how long the data written by the **DDSDataWriter** (p. 1113) is considered valid.

Classes

- struct **DDS_LifespanQosPolicy**

  *Specifies how long the data written by the **DDSDataWriter** (p. 1113) is considered valid.*

Variables

- const char *const **DDS_LIFESPAN_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_LifespanQosPolicy** (p. 773).*

5.103.1 Detailed Description

Specifies how long the data written by the **DDSDataWriter** (p. 1113) is considered valid.

5.103.2 Variable Documentation

5.103.2.1 const char* const DDS_LIFESPAN_QOS_POLICY_NAME

Stringified human-readable name for **DDS_LifespanQosPolicy** (p. 773).

5.104 WRITER_DATA_LIFECYCLE

Controls how a DataWriter handles the lifecycle of the instances (keys) that it is registered to manage.

Classes

- struct **DDS_WriterDataLifecycleQosPolicy**

  *Controls how a **DDSDataWriter** (p. 1113) handles the lifecycle of the instances (keys) that it is registered to manage.*

Variables

- const char *const **DDS_WRITERDATALIFECYCLE_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_WriterDataLifecycleQosPolicy** (p. 1071).*

5.104.1 Detailed Description

Controls how a DataWriter handles the lifecycle of the instances (keys) that it is registered to manage.

5.104.2 Variable Documentation

5.104.2.1 const char* const DDS_WRITERDATALIFECYCLE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_WriterDataLifecycleQosPolicy** (p. 1071).

5.105 READER_DATA_LIFECYCLE

Controls how a DataReader manages the lifecycle of the data that it has received.

Classes

- struct **DDS_ReaderDataLifecycleQosPolicy**

  Controls how a DataReader manages the lifecycle of the data that it has received.

Variables

- const char *const **DDS_READERDATALIFECYCLE_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_ReaderDataLifecycleQosPolicy*** (p. 859).

5.105.1 Detailed Description

Controls how a DataReader manages the lifecycle of the data that it has received.

5.105.2 Variable Documentation

5.105.2.1 const char* const DDS_READERDATALIFECYCLE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_ReaderDataLifecycleQosPolicy** (p. 859).

5.106 ENTITY_FACTORY

A QoS policy for all **DDSEntity** (p. 1253) types that can act as factories for one or more other **DDSEntity** (p. 1253) types.

Classes

- struct **DDS_EntityFactoryQosPolicy**

  *A QoS policy for all **DDSEntity** (p. 1253) types that can act as factories for one or more other **DDSEntity** (p. 1253) types.*

Variables

- const char *const **DDS_ENTITYFACTORY_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_EntityFactoryQosPolicy** (p. 733).*

5.106.1 Detailed Description

A QoS policy for all **DDSEntity** (p. 1253) types that can act as factories for one or more other **DDSEntity** (p. 1253) types.

5.106.2 Variable Documentation

5.106.2.1 const char* const DDS_ENTITYFACTORY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_EntityFactoryQosPolicy** (p. 733).

5.107 Extended Qos Support

*<<**eXtension**>>* *(p. 199)* Types and defines used in extended QoS policies.

Modules

- **Thread Settings**

  The properties of a thread of execution.

Classes

- struct **DDS_RtpsReliableReaderProtocol_t**

  Qos related to reliable reader protocol defined in RTPS.

- struct **DDS_RtpsReliableWriterProtocol_t**

  QoS related to the reliable writer protocol defined in RTPS.

5.107.1 Detailed Description

*<<**eXtension**>>* *(p. 199)* Types and defines used in extended QoS policies.

5.108 Unicast Settings

Unicast communication settings.

Classes

- struct **DDS_TransportUnicastSettings_t**

  Type representing a list of unicast locators.

- struct **DDS_TransportUnicastSettingsSeq**

  Declares IDL `sequence<` ***DDS_TransportUnicastSettings_t*** *(p. 989)* `>`.

5.108.1 Detailed Description

Unicast communication settings.

5.109 Multicast Settings

Multicast communication settings.

Classes

- struct **DDS_TransportMulticastSettings_t**

  Type representing a list of multicast locators.

- struct **DDS_TransportMulticastSettingsSeq**

  Declares IDL `sequence<` ***DDS_TransportMulticastSettings_t*** *(p. 980)* `>`.

5.109.1 Detailed Description

Multicast communication settings.

5.110 Multicast Mapping

Multicast communication mapping.

Classes

- struct **DDS_TransportMulticastMappingFunction_t**

  Type representing an external mapping function.

- struct **DDS_TransportMulticastMapping_t**

  Type representing a list of multicast mapping elements.

- struct **DDS_TransportMulticastMappingSeq**

  Declares IDL `sequence<` ***DDS_TransportMulticastMapping_t*** *(p. 971)* `>`.

5.110.1 Detailed Description

Multicast communication mapping.

5.111 TRANSPORT_SELECTION

<<*eXtension*>> *(p. 199)* Specifies the physical transports that a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087) may use to send or receive data.

Classes

- struct **DDS_TransportSelectionQosPolicy**

  *Specifies the physical transports a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087) may use to send or receive data.*

Variables

- const char *const **DDS_TRANSPORTSELECTION_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_TransportSelectionQosPolicy** (p. 985).*

5.111.1 Detailed Description

<<*eXtension*>> *(p. 199)* Specifies the physical transports that a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087) may use to send or receive data.

5.111.2 Variable Documentation

5.111.2.1 const char* const DDS_TRANSPORTSELECTION_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TransportSelectionQosPolicy** (p. 985).

5.112 TRANSPORT_UNICAST

<<*eXtension*>> *(*p. 199*)* Specifies a subset of transports and a port number that can be used by an Entity to receive data.

Modules

- **Unicast Settings**

  Unicast communication settings.

Classes

- struct **DDS_TransportUnicastQosPolicy**

  Specifies a subset of transports and a port number that can be used by an Entity to receive data.

Variables

- const char *const **DDS_TRANSPORTUNICAST_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_TransportUnicastQosPolicy*** *(*p. 987*)*.

5.112.1 Detailed Description

<<*eXtension*>> *(*p. 199*)* Specifies a subset of transports and a port number that can be used by an Entity to receive data.

5.112.2 Variable Documentation

5.112.2.1 const char* const DDS_TRANSPORTUNICAST_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TransportUnicastQosPolicy** (p. 987).

5.113 TRANSPORT_MULTICAST

<<*eXtension*>> *(p. 199)* Specifies the multicast address on which a **DDSDataReader** (p. 1087) wants to receive its data. It can also specify a port number, as well as a subset of the available (at the **DDSDomainParticipant** (p. 1139) level) transports with which to receive the multicast data.

Modules

- **Multicast Settings**

  Multicast communication settings.

- **Multicast Mapping**

  Multicast communication mapping.

Classes

- struct **DDS_TransportMulticastQosPolicy**

  *Specifies the multicast address on which a **DDSDataReader** (p. 1087) wants to receive its data. It can also specify a port number as well as a subset of the available (at the **DDSDomainParticipant** (p. 1139) level) transports with which to receive the multicast data.*

Enumerations

- enum **DDS_TransportMulticastQosPolicyKind** {
  DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS,
  DDS_UNICAST_ONLY_TRANSPORT_MULTICAST_QOS }

  Transport Multicast Policy Kind.

Variables

- const char *const **DDS_TRANSPORTMULTICAST_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_TransportMulticastQosPolicy** (p. 978).*

5.113.1 Detailed Description

<<*eXtension*>> *(p. 199)* Specifies the multicast address on which a **DDSDataReader** (p. 1087) wants to receive its data. It can also specify a port number, as well as a subset of the available (at the **DDSDomainParticipant** (p. 1139) level) transports with which to receive the multicast data.

5.113.2 Enumeration Type Documentation

5.113.2.1 enum DDS_TransportMulticastQosPolicyKind

Transport Multicast Policy Kind.

See also:

DDS_TransportMulticastQosPolicy (p. 978)

Enumerator:

DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS Selects the multicast address automatically.

NOTE: This setting is required when using the **DDS_TransportMulticastMappingQosPolicy** (p. 975).

DDS_UNICAST_ONLY_TRANSPORT_MULTICAST_QOS Selects a unicast-only mode.

5.113.3 Variable Documentation

5.113.3.1 const char* const DDS_TRANSPORTMULTICAST_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TransportMulticastQosPolicy** (p. 978).

5.114 TRANSPORT_MULTICAST_-MAPPING

<<*eXtension*>> *(*p. *199)* Specifies a list of topic expressions and addresses that can be used by an Entity with a specific topic name to receive data.

Classes

- struct **DDS_TransportMulticastMappingQosPolicy**

  Specifies a list of topic_expressions and multicast addresses that can be used by an Entity with a specific topic name to receive data.

Variables

- const char *const **DDS_TRANSPORTMULTICASTMAPPING_-QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_-TransportMulticastMappingQosPolicy*** *(*p. *975).*

5.114.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Specifies a list of topic expressions and addresses that can be used by an Entity with a specific topic name to receive data.

5.114.2 Variable Documentation

5.114.2.1 const char* const DDS_-TRANSPORTMULTICASTMAPPING_-QOS_POLICY_NAME

Stringified human-readable name for **DDS_-TransportMulticastMappingQosPolicy** (p. 975).

5.115 DISCOVERY

<<*eXtension*>> *(p. 199)* Specifies the attributes required to discover participants in the domain.

Modules

- **NDDS_DISCOVERY_PEERS**

  Environment variable or a file that specifies the default values of ***DDS_DiscoveryQosPolicy::initial_peers*** *(p. 583) and* ***DDS_DiscoveryQosPolicy::multicast_receive_addresses*** *(p. 584) contained in the* ***DDS_DomainParticipantQos::discovery*** *(p. 591) qos policy.*

Classes

- struct **DDS_DiscoveryQosPolicy**

  Configures the mechanism used by the middleware to automatically discover and connect with new remote applications.

Variables

- const char *const **DDS_DISCOVERY_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_DiscoveryQosPolicy*** *(p. 582).*

5.115.1 Detailed Description

<<*eXtension*>> *(p. 199)* Specifies the attributes required to discover participants in the domain.

5.115.2 Variable Documentation

5.115.2.1 const char* const DDS_DISCOVERY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DiscoveryQosPolicy** (p. 582).

5.116 NDDS_DISCOVERY_PEERS

Environment variable or a file that specifies the default values of **DDS_DiscoveryQosPolicy::initial_peers** (p. 583) and **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584) contained in the **DDS_DomainParticipantQos::discovery** (p. 591) qos policy.

The default value of the **DDS_DomainParticipantQos** (p. 588) is obtained by calling **DDSDomainParticipantFactory::get_default_participant_qos()** (p. 1224).

NDDS_DISCOVERY_PEERS specifies the default value of the **DDS_DiscoveryQosPolicy::initial_peers** (p. 583) and **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584) fields, when the default participant QoS policies have not been explictly set by the user (i.e., **DDSDomainParticipantFactory::set_default_participant_qos()** (p. 1222) has never been called or was called using **DDS_PARTICIPANT_QOS_DEFAULT** (p. 35)).

If NDDS_DISCOVERY_PEERS does *not* contain a multicast address, then the string sequence **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584) is cleared and the RTI discovery process will not listen for discovery messages via multicast.

If NDDS_DISCOVERY_PEERS contains one or more multicast addresses, the addresses will be stored in **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584), starting at element 0. They will be stored in the order in which they appear in NDDS_DISCOVERY_PEERS.

Note: IPv4 multicast addresses must have a prefix. Therefore, when using the UDPv6 transport: if there are any IPv4 multicast addresses in the peers list, make sure they have "udpv4://" in front of them (such as udpv4://239.255.0.1).

Note: Currently, RTI Connext will only listen for discovery traffic on the first multicast address (element 0) in **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584).

NDDS_DISCOVERY_PEERS provides a mechanism to dynamically switch the discovery configuration of an RTI Connext application without recompilation. The application programmer is free to not use the default values; instead use values supplied by other means.

NDDS_DISCOVERY_PEERS can be specified either in an environment variable as comma (',') separated `"peer descriptors"` (see **Peer Descriptor Format** (p. 389)) or in a file. These formats are described below.

5.116.1 Peer Descriptor Format

A **peer descriptor** string specifies a range of participants at a given `locator`. Peer descriptor strings are used in the **DDS_DiscoveryQosPolicy::initial_peers** (p. 583) field and the **DDSDomainParticipant::add_peer()** (p. 1199) operation.

The anatomy of a `peer` descriptor is illustrated below using a special "StarFabric" transport example.

A peer descriptor consists of:

optional **Participant ID**. If a simple integer is specified, it indicates the maximum participant ID to be contacted by the RTI Connext discovery mechanism at the given locator. If that integer is enclosed in square brackets (e.g.: [2]) *only* that Participant ID will be used. You can also specify a range in the form of [a,b]: in this case only the Participant IDs in that specific range are contacted. If omitted, a default value of 4 is implied.

ˆ **Locator**. See **Locator Format** (p. 389).

These are separated by the '@' character. The separator may be omitted if a participant ID limit is not explictly specified.

Note that the "participant ID limit" only applies to unicast locators; it is ignored for multicast locators (and therefore should be omitted for multicast peer descriptors).

5.116.1.1 Locator Format

A **locator** string specifies a transport and an address in string format. Locators are used to form peer descriptors. A locator is equivalent to a peer descriptor with the default maximum participant ID.

A locator consists of:

optional **Transport name** (**alias** or **class**). This identifies the set of transport plugins (**Transport Aliases** (p. 125)) that may be used to parse the `address` portion of the locator. Note that a transport class name is an implicit alias that is used to refer to all the transport plugin instances of that class.

optional **Address**. See **Address Format** (p. 390).

These are separated by the "//:" string. The separator is specified if and only if a transport name is specified.

If a transport name is specified, the address may be omitted; in that case, all the unicast addresses (across all transport plugin instances) associated with the transport class are implied. Thus, a locator string may specify several addresses.

If an address is specified, the transport name and the separator string may be omitted; in that case all the available transport plugins (for the **DDSEntity** (p. 1253)) may be used to parse the address string.

5.116.1.2 Address Format

An **address** string specifies a transport-independent network address that qualifies a **transport-dependent** address string. Addresses are used to form locators. Addresses are also used in **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584), and **DDS_TransportMulticastSettings_t::receive_address** (p. 981) fields. An address is equivalent to a locator in which the transport name and separator are omitted.

An address consists of:

optional **Network Address**. An address in IPv4 or IPv6 string notation. If omitted, the network address of the transport is implied (**Transport Network Address** (p. 128)).

optional **Transport Address**. A string that is passed to the transport for processing. The transport maps this string into **NDDS_Transport_Property_t::address_bit_count** (p. 1524) bits. If omitted the network address is used as the fully qualified address.

These are separated by the '#' character. If a separator is specified, it must be followed by a non-empty string which is passed to the transport plugin.

The bits resulting from the transport address string are prepended with the network address. The least significant **NDDS_Transport_Property_t::address_bit_count** (p. 1524) bits of the network address are ignored (**Transport Network Address** (p. 128)).

If the separator is omitted and the string is not a valid IPv4 or IPv6 address, it is treated as a transport address with an implicit network address (of the transport plugin).

5.116.2 NDDS_DISCOVERY_PEERS Environment Variable Format

NDDS_DISCOVERY_PEERS can be specified via an environment variable of the same name, consisting of a sequence of peer descriptors separated by the comma (',') character.

Examples

Multicast (maximum participant ID is irrelevant)

- 239.255.0.1

Default maximum participant ID on localhost

- localhost

Default maximum participant ID on host 192.168.1.1 (IPv4)

- 192.168.1.1

Default maximum participant ID on host FAA0::0 (IPv6)

- FAA0::1

Default maximum participant ID on host FAA0::0#localhost (could be a UDPv4 transport plugin registered at network address of FAA0::0) (IPv6)

- FAA0::0#localhost

Default maximum participant ID on host himalaya accessed using the "udpv4" transport plugin(s) (IPv4)

- udpv4://himalaya

Default maximum participant ID on localhost using the "udpv4" transport plugin(s) registered at network address FAA0::0

- udpv4://FAA0::0#localhost

Default maximum participant ID on all unicast addresses accessed via the "udpv4" (UDPv4) transport plugin(s)

- udpv4://

Default maximum participant ID on host 0/0/R (StarFabric)

- 0/0/R
- #0/0/R

Default maximum participant ID on host 0/0/R (StarFabric) using the "starfabric" (StarFabric) transport plugin(s)

- starfabric://0/0/R
- starfabric://#0/0/R

Default maximum participant ID on host 0/0/R (StarFabric) using the "starfabric" (StarFabric) transport plugin(s) registered at network address FAA0::0

- starfabric://FBB0::0#0/0/R

Default maximum participant ID on all unicast addresses accessed via the "starfabric" (StarFabric) transport plugin(s)

- starfabric://

Default maximum participant ID on all unicast addresses accessed via the "shmem" (shared memory) transport plugin(s)

- shmem://

Default maximum participant ID on all unicast addresses accessed via the "shmem" (shared memory) transport plugin(s) registered at network address FCC0::0

- shmem://FCC0::0

Default maximum participant ID on hosts himalaya and gangotri

- himalaya,gangotri

Maximum participant ID of 1 on hosts himalaya and gangotri

- 1@himalaya,1@gangotri

Combinations of above

- 239.255.0.1,localhost,192.168.1.1,0/0/R
- FAA0::1,FAA0::0#localhost,FBB0::0#0/0/R
- udpv4://himalaya,udpv4://FAA0::0#localhost,#0/0/R
- starfabric://0/0/R,starfabric://FBB0::0#0/0/R,shmem://
- starfabric://,shmem://FCC0::0,1@himalaya,1@gangotri

5.116.3 NDDS_DISCOVERY_PEERS File Format

NDDS_DISCOVERY_PEERS can be specified via a file of the same name in the program's current working directory. A NDDS_DISCOVERY_PEERS file would contain a sequence of peer descriptors separated by whitespace or the comma (',') character. The file may also contain comments starting with a semicolon (';') character till the end of the line.

Example:

```
;; NDDS_DISCOVERY_PEERS - Default Discovery Configuration File
;;
;;
;; NOTE:
;;   1. This file must be in the current working directory, i.e.
;;      in the folder from which the application is launched.
;;
;;   2. This file takes precedence over the environment variable NDDS_DISCOVERY_PEERS
;;

;; Multicast
239.255.0.1                 ; The default RTI Connext discovery multicast address

;; Unicast
localhost,192.168.1.1       ; A comma can be used a separator

FAA0::1  FAA0::0#localhost  ; Whitespace can be used as a separator

1@himalaya                  ; Maximum participant ID of 1 on 'himalaya'
1@gangotri


;; UDPv4
udpv4://himalaya            ; 'himalaya' via 'udpv4' transport plugin(s)
udpv4://FAA0::0#localhost   ; 'localhost' via 'updv4' transport
                            ;  plugin registered at network address FAA0::0

;; Shared Memory
shmem://                    ; All 'shmem' transport plugin(s)
builtin.shmem://            ; The builtin 'shmem' transport plugin
shmem://FCC0::0             ; Shared memory transport plugin registered
                            ;                at network address FCC0::0

;; StarFabric
0/0/R                       ; StarFabric node 0/0/R
starfabric://0/0/R          ; 0/0/R accessed via 'starfabric'
                            ;             transport plugin(s)
starfabric://FBB0::0#0/0/R  ; StarFabric transport plugin registered
                            ;             at network address FBB0::0
starfabric://               ; All 'starfabric' transport plugin(s)
```

5.116.4 NDDS_DISCOVERY_PEERS Precedence

If the current working directory from which the RTI Connext application is launched contains a file called NDDS_DISCOVERY_PEERS, and an environment variable named NDDS_DISCOVERY_PEERS is also defined, the file takes precedence; the environment variable is ignored.

5.116.5 NDDS_DISCOVERY_PEERS Default Value

If NDDS_DISCOVERY_PEERS is not specified (either as a file in the current working directory, or as an environment variable), it implicitly defaults to the following.

```
;; Multicast (only on platforms which allow UDPv4 multicast out of the box)
;;
;; This allows any RTI Connext applications anywhere on the local network to
;; discover each other over UDPv4.

builtin.udpv4://239.255.0.1  ; RTI Connext's default discovery multicast address


;; Unicast - UDPv4 (on all platforms)
;;
;; This allows two RTI Connext applications using participant IDs up to the maximum
;; default participant ID on the local host and domain to discover each
;; other over UDP/IPv4.

builtin.udpv4://127.0.0.1


;; Unicast - Shared Memory (only on platforms that support shared memory)
;;
;; This allows two RTI Connext applications using participant IDs up to the maximum
;; default participant ID on the local host and domain to discover each
;; other over shared memory.

builtin.shmem://
```

5.116.6 Builtin Transport Class Names

The class names for the builtin transport plugins are:

- `shmem` - ::**Shared Memory Transport** (p. 257)
- `udpv4` - ::**UDPv4 Transport** (p. 265)
- `udpv6` - ::**UDPv6 Transport** (p. 275)

These may be used as the transport names in the **Locator Format** (p. 389).

5.116.7 NDDS_DISCOVERY_PEERS and Local Host Communication

Suppose you want to communicate with other RTI Connext applications on the same host and you are setting NDDS_DISCOVERY_PEERS explicitly (generally in order to use unicast discovery with applications on other hosts).

If the local host platform does not support the shared memory transport, then you can include the name of the local host in the NDDS_DISCOVERY_PEERS list.

If the local host platform supports the shared memory transport, then you can do one of the following:

- Include "shmem://" in the NDDS_DISCOVERY_PEERS list. This will cause shared memory to be used for discovery and data traffic for applications on the same host.

or:

- Include the name of the local host in the NDDS_DISCOVERY_PEERS list and disable the shared memory transport in the **DDS_TransportBuiltinQosPolicy** (p. 969) of the **DDSDomainParticipant** (p. 1139). This will cause UDP loopback to be used for discovery and data traffic for applications on the same host.

(To check if your platform supports shared memory, see the `Platform Notes`.)

See also:

DDS_DiscoveryQosPolicy::multicast_receive_addresses (p. 584)
DDS_DiscoveryQosPolicy::initial_peers (p. 583)
DDSDomainParticipant::add_peer() (p. 1199)
DDS_PARTICIPANT_QOS_DEFAULT (p. 35)
DDSDomainParticipantFactory::get_default_participant_qos() (p. 1224)
Transport Aliases (p. 125)
Transport Network Address (p. 128)
NDDSTransportSupport::register_transport() (p. 1560)

5.117 TRANSPORT_BUILTIN

<<*eXtension*>> *(*p. *199)* Specifies which built-in transports are used.

Classes

ˆ struct **DDS_TransportBuiltinQosPolicy**

Specifies which built-in transports are used.

Defines

ˆ #define **DDS_TRANSPORTBUILTIN_MASK_NONE**

None of the built-in transports will be registered automatically when the ***DDSDomainParticipant*** *(*p. *1139) is enabled. The user must explictly register transports using* ***NDDSTransportSupport::register_transport*** *(*p. *1560).*

ˆ #define **DDS_TRANSPORTBUILTIN_MASK_DEFAULT**

The default value of ***DDS_TransportBuiltinQosPolicy::mask*** *(*p. *970).*

ˆ #define **DDS_TRANSPORTBUILTIN_MASK_ALL**

All the available built-in transports are registered automatically when the ***DDSDomainParticipant*** *(*p. *1139) is enabled.*

Typedefs

ˆ typedef **DDS_Long DDS_TransportBuiltinKindMask**

A mask of ***DDS_TransportBuiltinKind*** *(*p. *398) bits.*

Enumerations

ˆ enum **DDS_TransportBuiltinKind** {
DDS_TRANSPORTBUILTIN_UDPv4,
DDS_TRANSPORTBUILTIN_SHMEM ,
DDS_TRANSPORTBUILTIN_UDPv6 }

Built-in transport kind.

Variables

- const char *const **DDS_TRANSPORTBUILTIN_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_TransportBuiltinQosPolicy** (p. 969).*

- const char *const **DDS_TRANSPORTBUILTIN_SHMEM_ALIAS**

  Alias name for the shared memory built-in transport.

- const char *const **DDS_TRANSPORTBUILTIN_UDPv4_ALIAS**

  Alias name for the UDPv4 built-in transport.

- const char *const **DDS_TRANSPORTBUILTIN_UDPv6_ALIAS**

  Alias name for the UDPv6 built-in transport.

5.117.1 Detailed Description

<<***eXtension***>> *(p. 199)* Specifies which built-in transports are used.

See also:

Changing the automatically registered built-in transports (p. 181)

5.117.2 Define Documentation

5.117.2.1 #define DDS_TRANSPORTBUILTIN_MASK_NONE

None of the built-in transports will be registered automatically when the **DDSDomainParticipant** (p. 1139) is enabled. The user must explictly register transports using **NDDSTransportSupport::register_transport** (p. 1560).

See also:

DDS_TransportBuiltinKindMask (p. 398)

5.117.2.2 #define DDS_TRANSPORTBUILTIN_MASK_DEFAULT

The default value of **DDS_TransportBuiltinQosPolicy::mask** (p. 970).

The set of builtin transport plugins that will be automatically registered with the participant by default. The user can register additional transports using **NDDSTransportSupport::register_transport** (p. 1560).

See also:

DDS_TransportBuiltinKindMask (p. 398)

5.117.2.3 #define DDS_TRANSPORTBUILTIN_MASK_ALL

All the available built-in transports are registered automatically when the **DDS-DomainParticipant** (p. 1139) is enabled.

See also:

DDS_TransportBuiltinKindMask (p. 398)

5.117.3 Typedef Documentation

5.117.3.1 typedef DDS_Long DDS_TransportBuiltinKindMask

A mask of **DDS_TransportBuiltinKind** (p. 398) bits.

QoS:

DDS_TransportBuiltinQosPolicy (p. 969)

5.117.4 Enumeration Type Documentation

5.117.4.1 enum DDS_TransportBuiltinKind

Built-in transport kind.

See also:

DDS_TransportBuiltinKindMask (p. 398)

Enumerator:

DDS_TRANSPORTBUILTIN_UDPv4 Built-in UDPv4 transport, ::**UDPv4 Transport** (p. 265).

DDS_TRANSPORTBUILTIN_SHMEM Built-in shared memory transport, ::**Shared Memory Transport** (p. 257).

DDS_TRANSPORTBUILTIN_UDPv6 Built-in UDPv6 transport, ::**UDPv6 Transport** (p. 275).

5.117.5 Variable Documentation

5.117.5.1 const char* const DDS_TRANSPORTBUILTIN_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TransportBuiltinQosPolicy** (p. 969).

5.117.5.2 const char* const DDS_TRANSPORTBUILTIN_SHMEM_ALIAS

Alias name for the shared memory built-in transport.

5.117.5.3 const char* const DDS_TRANSPORTBUILTIN_UDPv4_ALIAS

Alias name for the UDPv4 built-in transport.

5.117.5.4 const char* const DDS_TRANSPORTBUILTIN_UDPv6_ALIAS

Alias name for the UDPv6 built-in transport.

5.118 WIRE_PROTOCOL

*<<**eXtension**>>* *(p. 199)* Specifies the wire protocol related attributes for the **DDSDomainParticipant** (p. 1139).

Classes

- struct **DDS_RtpsWellKnownPorts_t**

  RTPS well-known port mapping configuration.

- struct **DDS_WireProtocolQosPolicy**

  *Specifies the wire-protocol-related attributes for the **DDSDomainParticipant** (p. 1139).*

Defines

- #define **DDS_RTPS_RESERVED_PORT_MASK_DEFAULT**

  *The default value of **DDS_WireProtocolQosPolicy::rtps_reserved_port_mask** (p. 1065).*

- #define **DDS_RTPS_RESERVED_PORT_MASK_NONE**

  No bits are set.

- #define **DDS_RTPS_RESERVED_PORT_MASK_ALL**

  All bits are set.

Typedefs

- typedef **DDS_Long DDS_RtpsReservedPortKindMask**

  *A mask of **DDS_RtpsReservedPortKind** (p. 403) bits.*

Enumerations

- enum **DDS_RtpsReservedPortKind** {

  DDS_RTPS_RESERVED_PORT_BUILTIN_UNICAST = 0x0001 << 0,

  DDS_RTPS_RESERVED_PORT_BUILTIN_MULTICAST = 0x0001 << 1,

DDS_RTPS_RESERVED_PORT_USER_UNICAST = 0x0001 << 2,

DDS_RTPS_RESERVED_PORT_USER_MULTICAST = 0x0001 << 3 }

RTPS reserved port kind, used to identify the types of ports that can be reserved on domain participant enable.

ˆ enum **DDS_WireProtocolQosPolicyAutoKind** {

DDS_RTPS_AUTO_ID_FROM_IP,

DDS_RTPS_AUTO_ID_FROM_MAC }

Kind of auto mechanism used to calculate the GUID prefix.

Variables

ˆ struct **DDS_RtpsWellKnownPorts_t DDS_RTI_BACKWARDS_COMPATIBLE_RTPS_WELL_KNOWN_PORTS**

Assign to use well-known port mappings which are compatible with previous versions of the RTI Connext middleware.

ˆ struct **DDS_RtpsWellKnownPorts_t DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS**

Assign to use well-known port mappings which are compliant with OMG's DDS Interoperability Wire Protocol.

ˆ const char *const **DDS_WIREPROTOCOL_QOS_POLICY_NAME**

*Stringified human-readable name for **DDS_WireProtocolQosPolicy** (p. 1059).*

5.118.1 Detailed Description

<<***eXtension***>> *(p. 199)* Specifies the wire protocol related attributes for the **DDSDomainParticipant** (p. 1139).

5.118.2 Define Documentation

5.118.2.1 #define DDS_RTPS_RESERVED_PORT_MASK_DEFAULT

Value:

```
((DDS_RtpsReservedPortKindMask) DDS_RTPS_RESERVED_PORT_BUILTIN_UNICAST \
    | DDS_RTPS_RESERVED_PORT_BUILTIN_MULTICAST | DDS_RTPS_RESERVED_PORT_USER_UNICAST)
```

The default value of **DDS_WireProtocolQosPolicy::rtps_reserved_port_mask** (p. 1065).

Most of the ports that may be needed by DDS will be reserved by the transport when the participant is enabled. With this value set, failure to allocate a port that is computed based on the **DDS_RtpsWellKnownPorts_t** (p. 905) will be detected at this time and the enable operation will fail.

This setting will avoid reserving the **usertraffic** multicast port, which is not actually used unless there are DataReaders that enable multicast but fail to specify a port.

Automatic participant ID selection will be based on finding a participant index with both the discovery (metatraffic) unicast port and usertraffic unicast port available.

See also:

DDS_RtpsReservedPortKindMask (p. 403)

5.118.2.2 #define DDS_RTPS_RESERVED_PORT_MASK_NONE

No bits are set.

None of the ports that are needed by DDS will be allocated until they are specifically required. With this value set, automatic participant Id selection will be based on selecting a port for discovery (metatraffic) unicast traffic on a single transport.

See also:

DDS_RtpsReservedPortKindMask (p. 403)

5.118.2.3 #define DDS_RTPS_RESERVED_PORT_MASK_ALL

All bits are set.

All of the ports that may be needed by DDS will be reserved when the participant is enabled. With this value set, failure to allocate a port that is computed based on the **DDS_RtpsWellKnownPorts_t** (p. 905) will be detected at this time, and the enable operation will fail.

Note that this will also reserve the **usertraffic** multicast port which is not actually used unless there are DataReaders that enable multicast but fail to

specify a port. To avoid unnecesary resource usage for these ports, use RTPS_-RESERVED_PORT_MASK_DEFAULT.

Automatic participant ID selection will be based on finding a participant index with both the discovery (metatraffic) unicast port and usertraffic unicast port available.

See also:

DDS_RtpsReservedPortKindMask (p. 403)

5.118.3 Typedef Documentation

5.118.3.1 typedef DDS_Long DDS_RtpsReservedPortKindMask

A mask of **DDS_RtpsReservedPortKind** (p. 403) bits.

QoS:

DDS_WireProtocolQosPolicy (p. 1059)

5.118.4 Enumeration Type Documentation

5.118.4.1 enum DDS_RtpsReservedPortKind

RTPS reserved port kind, used to identify the types of ports that can be reserved on domain participant enable.

See also:

DDS_WireProtocolQosPolicy::rtps_reserved_port_mask (p. 1065)

Enumerator:

DDS_RTPS_RESERVED_PORT_BUILTIN_UNICAST Select the **metatraffic** unicast port.

DDS_RTPS_RESERVED_PORT_BUILTIN_MULTICAST Select the **metatraffic** multicast port.

DDS_RTPS_RESERVED_PORT_USER_UNICAST Select the **usertraffic** unicast port.

DDS_RTPS_RESERVED_PORT_USER_MULTICAST Select the **usertraffic** multicast port.

5.118.4.2 enum DDS_WireProtocolQosPolicyAutoKind

Kind of auto mechanism used to calculate the GUID prefix.

See also:

DDS_WireProtocolQosPolicy::rtps_auto_id_kind (p. 1066)

Enumerator:

DDS_RTPS_AUTO_ID_FROM_IP Select the **IPv4** based algorithm.

DDS_RTPS_AUTO_ID_FROM_MAC Select the **MAC** based algorithm.

Note to Solaris Users: To use DDS_RTPS_AUTO_ID_FROM_MAC, you must run the RTI Connext application while logged in as 'root.'

5.118.5 Variable Documentation

5.118.5.1 struct DDS_RtpsWellKnownPorts_t DDS_RTI_BACKWARDS_COMPATIBLE_RTPS_WELL_KNOWN_PORTS

Assign to use well-known port mappings which are compatible with previous versions of the RTI Connext middleware.

Assign **DDS_WireProtocolQosPolicy::rtps_well_known_ports** (p. 1065) to this value to remain compatible with previous versions of the RTI Connext middleware that used fixed port mappings.

The following are the `rtps_well_known_ports` values for **DDS_RTI_BACKWARDS_COMPATIBLE_RTPS_WELL_KNOWN_PORTS** (p. 404):

`port_base` = 7400

`domain_id_gain` = 10

`participant_id_gain` = 1000

`builtin_multicast_port_offset` = 2

`builtin_unicast_port_offset` = 0

`user_multicast_port_offset` = 1

`user_unicast_port_offset` = 3

These settings are *not* compliant with OMG's DDS Interoperability Wire Protocol. To comply with the specification, please use **DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS** (p. 405).

See also:

DDS_WireProtocolQosPolicy::rtps_well_known_ports (p. 1065)
DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS (p. 405)

5.118.5.2 struct DDS_RtpsWellKnownPorts_t DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS

Assign to use well-known port mappings which are compliant with OMG's DDS Interoperability Wire Protocol.

Assign **DDS_WireProtocolQosPolicy::rtps_well_known_ports** (p. 1065) to this value to use well-known port mappings which are compliant with OMG's DDS Interoperability Wire Protocol.

The following are the `rtps_well_known_ports` values for **DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS** (p. 405):

`port_base` = 7400

`domain_id_gain` = 250

`participant_id_gain` = 2

`builtin_multicast_port_offset` = 0

`builtin_unicast_port_offset` = 10

`user_multicast_port_offset` = 1

`user_unicast_port_offset` = 11

Assuming a maximum port number of 65535 (UDPv4), the above settings enable the use of about 230 domains with up to 120 Participants per node per domain.

These settings are *not* backwards compatible with previous versions of the RTI Connext middleware that used fixed port mappings. For backwards compability, please use **DDS_RTI_BACKWARDS_COMPATIBLE_RTPS_WELL_KNOWN_PORTS** (p. 404).

See also:

DDS_WireProtocolQosPolicy::rtps_well_known_ports (p. 1065)
DDS_RTI_BACKWARDS_COMPATIBLE_RTPS_WELL_KNOWN_PORTS (p. 404)

5.118.5.3 const char* const DDS_WIREPROTOCOL_QOS_POLICY_NAME

Stringified human-readable name for **DDS_WireProtocolQosPolicy** (p. 1059).

5.119 DATA_READER_RESOURCE_LIMITS

<<***eXtension***>> *(p. 199)* Various settings that configure how DataReaders allocate and use physical memory for internal resources.

Classes

- struct **DDS_DataReaderResourceLimitsQosPolicy**

  Various settings that configure how a ***DDSDataReader*** *(p. 1087) allocates and uses physical memory for internal resources.*

Variables

- const char *const **DDS_DATAREADERRESOURCELIMITS_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_DataReaderResourceLimitsQosPolicy*** *(p. 521).*

- const **DDS_Long DDS_AUTO_MAX_TOTAL_INSTANCES**

  <<**eXtension**>> (p. 199) *This value is used to make* ***DDS_DataReaderResourceLimitsQosPolicy::max_total_instances*** *(p. 530) equal to* ***DDS_ResourceLimitsQosPolicy::max_instances*** *(p. 882).*

5.119.1 Detailed Description

<<***eXtension***>> *(p. 199)* Various settings that configure how DataReaders allocate and use physical memory for internal resources.

5.119.2 Variable Documentation

5.119.2.1 const char* const DDS_DATAREADERRESOURCELIMITS_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DataReaderResourceLimitsQosPolicy** (p. 521).

5.119.2.2 const DDS_Long DDS_AUTO_MAX_TOTAL_INSTANCES

<<*eXtension*>> *(p. 199)* This value is used to make **DDS_DataReaderResourceLimitsQosPolicy::max_total_instances** (p. 530) equal to **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882).

5.120 DATA_WRITER_RESOURCE_LIMITS

<<eXtension>> *(p. 199)* Various settings that configure how a **DDS-DataWriter** (p. 1113) allocates and uses physical memory for internal resources.

Classes

- struct **DDS_DataWriterResourceLimitsQosPolicy**

  *Various settings that configure how a **DDSDataWriter** (p. 1113) allocates and uses physical memory for internal resources.*

Enumerations

- enum **DDS_DataWriterResourceLimitsInstanceReplacementKind** {

  DDS_UNREGISTERED_INSTANCE_REPLACEMENT,

  DDS_ALIVE_INSTANCE_REPLACEMENT,

  DDS_DISPOSED_INSTANCE_REPLACEMENT,

  DDS_ALIVE_THEN_DISPOSED_INSTANCE_-REPLACEMENT,

  DDS_DISPOSED_THEN_ALIVE_INSTANCE_-REPLACEMENT,

  DDS_ALIVE_OR_DISPOSED_INSTANCE_REPLACEMENT
  }

  Sets the kinds of instances that can be replaced when instance resource limits are reached.

Variables

- const char *const **DDS_DATAWRITERRESOURCELIMITS_-QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_-DataWriterResourceLimitsQosPolicy** (p. 560).*

5.120.1 Detailed Description

<<eXtension>> *(p. 199)* Various settings that configure how a **DDSDataWriter** (p. 1113) allocates and uses physical memory for internal resources.

5.120.2 Enumeration Type Documentation

5.120.2.1 enum DDS_DataWriterResourceLimitsInstanceReplacementKind

Sets the kinds of instances that can be replaced when instance resource limits are reached.

When **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882) is reached, a **DDSDataWriter** (p. 1113) will try to make room for a new instance by attempting to reclaim an existing instance based on the instance replacement kind specified by **DDS_DataWriterResourceLimitsQosPolicy::instance_replacement** (p. 563).

Only instances whose states match the specified kinds are eligible to be replaced. In addition, an instance must have had all of its samples fully acknowledged for it to be considered replaceable.

For all kinds, a **DDSDataWriter** (p. 1113) will replace the oldest instance satisfying that kind. For example, when the kind is **DDS_UNREGISTERED_INSTANCE_REPLACEMENT** (p. 411), a **DDSDataWriter** (p. 1113) will remove the oldest fully acknowledged unregistered instance, if such an instance exists.

If no replaceable instance exists, the invoked function will either return with an appropriate out-of-resources return code, or in the case of a write, it may first block to wait for an instance to be acknowledged. Otherwise, the **DDSDataWriter** (p. 1113) will replace the old instance with the new instance, and invoke, if available, the DDS_DataWriterListener_InstanceReplacedCallback to notify the user about an instance being replaced.

A **DDSDataWriter** (p. 1113) checks for replaceable instances in the following order, stopping once a replaceable instance is found:

If **DDS_DataWriterResourceLimitsQosPolicy::replace_empty_instances** (p. 564) is **DDS_BOOLEAN_TRUE** (p. 298), a **DDSDataWriter** (p. 1113) first tries replacing instances that have no samples. These empty instances can be unregistered, disposed, or alive. Next, a **DDSDataWriter** (p. 1113) tries replacing unregistered instances. Since an unregistered instance indicates that the **DDSDataWriter** (p. 1113) is done modifying it, unregistered instances are replaced before instances of any other state (alive, disposed). This is the same as the **DDS_**

UNREGISTERED_INSTANCE_REPLACEMENT (p. 411) kind. Then, a **DDSDataWriter** (p. 1113) tries replacing what is specified by **DDS_DataWriterResourceLimitsQosPolicy::instance_replacement** (p. 563). With unregistered instances already checked, this leaves alive and disposed instances. When both alive and disposed instances may be replaced, the kind specifies whether the particular order matters (e.g. DISPOSED_THEN_ALIVE, ALIVE_THEN_DISPOSED) or not (ALIVE_OR_DISPOSED).

QoS:

DDS_DataWriterResourceLimitsQosPolicy (p. 560)

Enumerator:

DDS_UNREGISTERED_INSTANCE_REPLACEMENT Allows a **DDSDataWriter** (p. 1113) to reclaim unregistered acknowledged instances.

By default all instance replacement kinds first attempt to reclaim an unregistered acknowledged instance. Used in **DDS_DataWriterResourceLimitsQosPolicy::instance_replacement** (p. 563) **[default]**

DDS_ALIVE_INSTANCE_REPLACEMENT Allows a **DDSDataWriter** (p. 1113) to reclaim alive acknowledged instances.

When an unregistered acknowledged instance is not available to reclaim, this kind allows a **DDSDataWriter** (p. 1113) to reclaim an alive acknowledged instance, where an alive instance is a registered, non-disposed instance. The least recently registered or written alive instance will be reclaimed.

DDS_DISPOSED_INSTANCE_REPLACEMENT Allows a **DDSDataWriter** (p. 1113) to reclaim disposed acknowledged instances.

When an unregistered acknowledged instance is not available to reclaim, this kind allows a **DDSDataWriter** (p. 1113) to reclaim a disposed acknowledged instance. The least recently disposed instance will be reclaimed.

DDS_ALIVE_THEN_DISPOSED_INSTANCE_REPLACEMENT Allows a **DDSDataWriter** (p. 1113) first to reclaim an alive acknowledged instance, and then if necessary a disposed acknowledged instance.

When an unregistered acknowledged instance is not available to reclaim, this kind allows a **DDSDataWriter** (p. 1113) first try reclaiming an alive acknowledged instance. If no instance is reclaimable, then it trys reclaiming a disposed acknowledged instance. The least recently used (i.e. registered, written, or disposed) instance will be reclaimed.

DDS_DISPOSED_THEN_ALIVE_INSTANCE_REPLACEMENT
Allows a **DDSDataWriter** (p. 1113) first to reclaim a disposed acknowledged instance, and then if necessary an alive acknowledged instance.

When an unregistered acknowledged instance is not available to reclaim, this kind allows a **DDSDataWriter** (p. 1113) first try reclaiming a disposed acknowledged instance. If no instance is reclaimable, then it trys reclaiming an alive acknowledged instance. The least recently used (i.e. disposed, registered, or written) instance will be reclaimed.

DDS_ALIVE_OR_DISPOSED_INSTANCE_REPLACEMENT
Allows a **DDSDataWriter** (p. 1113) to reclaim a either an alive acknowledged instance or a disposed acknowledged instance.

When an unregistered acknowledged instance is not available to reclaim, this kind allows a **DDSDataWriter** (p. 1113) to reclaim either an alive acknowledged instance or a disposed acknowledged instance. If both instance kinds are available to reclaim, the **DDSDataWriter** (p. 1113) will reclaim the least recently used (i.e. disposed, registered, or written) instance.

5.120.3 Variable Documentation

5.120.3.1 const char* const DDS_DATAWRITERRESOURCELIMITS_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DataWriterResourceLimitsQosPolicy** (p. 560).

5.121 DATA_READER_PROTOCOL

<<*eXtension*>> *(*p. *199)* Specifies the DataReader-specific protocol QoS.

Classes

- struct **DDS_DataReaderProtocolQosPolicy**

  Along with ***DDS_WireProtocolQosPolicy*** *(*p. *1059) and* ***DDS_DataWriterProtocolQosPolicy*** *(*p. *535), this QoS policy configures the DDS on-the-network protocol (RTPS).*

Variables

- const char *const **DDS_DATAREADERPROTOCOL_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_DataReaderProtocolQosPolicy*** *(*p. *501).*

5.121.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Specifies the DataReader-specific protocol QoS.

5.121.2 Variable Documentation

5.121.2.1 const char* const DDS_DATAREADERPROTOCOL_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DataReaderProtocolQosPolicy** (p. 501).

5.122 DATA_WRITER_PROTOCOL

<<*eXtension*>> *(p. 199)* Along with **DDS_WireProtocolQosPolicy** (p. 1059) and **DDS_DataReaderProtocolQosPolicy** (p. 501), this QoS policy configures the DDS on-the-network protocol (RTPS).

Classes

- struct **DDS_DataWriterProtocolQosPolicy**

  Protocol that applies only to ***DDSDataWriter*** *(p. 1113) instances.*

Variables

- const char *const **DDS_DATAWRITERPROTOCOL_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_DataWriterProtocolQosPolicy*** *(p. 535).*

5.122.1 Detailed Description

<<*eXtension*>> *(p. 199)* Along with **DDS_WireProtocolQosPolicy** (p. 1059) and **DDS_DataReaderProtocolQosPolicy** (p. 501), this QoS policy configures the DDS on-the-network protocol (RTPS).

5.122.2 Variable Documentation

5.122.2.1 const char* const DDS_DATAWRITERPROTOCOL_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DataWriterProtocolQosPolicy** (p. 535).

5.123 SYSTEM_RESOURCE_LIMITS

<<*eXtension*>> *(p. 199)* Configures DomainParticipant-independent resources used by RTI Connext.

Classes

- struct **DDS_SystemResourceLimitsQosPolicy**

  *Configures **DDSDomainParticipant** (p. 1139)-independent resources used by RTI Connext. Mainly used to change the maximum number of **DDSDomainParticipant** (p. 1139) entities that can be created within a single process (address space).*

Variables

- const char *const **DDS_SYSTEMRESOURCELIMITS_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_SystemResourceLimitsQosPolicy** (p. 948).*

5.123.1 Detailed Description

<<*eXtension*>> *(p. 199)* Configures DomainParticipant-independent resources used by RTI Connext.

5.123.2 Variable Documentation

5.123.2.1 const char* const DDS_SYSTEMRESOURCELIMITS_QOS_POLICY_NAME

Stringified human-readable name for **DDS_SystemResourceLimitsQosPolicy** (p. 948).

5.124 DOMAIN_PARTICIPANT_-RESOURCE_LIMITS

<<eXtension>> *(p. 199)* Various settings that configure how a **DDSDomainParticipant** (p. 1139) allocates and uses physical memory for internal resources, including the maximum sizes of various properties.

Classes

- struct **DDS_AllocationSettings_t**

  Resource allocation settings.

- struct **DDS_DomainParticipantResourceLimitsQosPolicy**

  *Various settings that configure how a **DDSDomainParticipant** (p. 1139) allocates and uses physical memory for internal resources, including the maximum sizes of various properties.*

Variables

- const char *const **DDS_DOMAINPARTICIPANTRESOURCELIMITS_-QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_-DomainParticipantResourceLimitsQosPolicy** (p. 593).*

5.124.1 Detailed Description

<<eXtension>> *(p. 199)* Various settings that configure how a **DDSDomainParticipant** (p. 1139) allocates and uses physical memory for internal resources, including the maximum sizes of various properties.

5.124.2 Variable Documentation

5.124.2.1 const char* const DDS_-DOMAINPARTICIPANTRESOURCELIMITS_QOS_-POLICY_NAME

Stringified human-readable name for **DDS_-DomainParticipantResourceLimitsQosPolicy** (p. 593).

5.125 EVENT

<<*eXtension*>> *(*p. *199)* Configures the internal thread in a DomainParticipant that handles timed events.

Classes

- struct **DDS_EventQosPolicy**

  Settings for event.

Variables

- const char *const **DDS_EVENT_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_EventQosPolicy*** *(*p. *739).*

5.125.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Configures the internal thread in a DomainParticipant that handles timed events.

5.125.2 Variable Documentation

5.125.2.1 const char* const DDS_EVENT_QOS_POLICY_NAME

Stringified human-readable name for **DDS_EventQosPolicy** (p. 739).

5.126 DATABASE

<<*eXtension*>> *(*p. *199)* Various threads and resource limits settings used by RTI Connext to control its internal database.

Classes

- struct **DDS_DatabaseQosPolicy**

  Various threads and resource limits settings used by RTI Connext to control its internal database.

Variables

- const char *const **DDS_DATABASE_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_DatabaseQosPolicy*** *(*p. *495).*

5.126.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Various threads and resource limits settings used by RTI Connext to control its internal database.

5.126.2 Variable Documentation

5.126.2.1 const char* const DDS_DATABASE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DatabaseQosPolicy** (p. 495).

5.127 RECEIVER_POOL

<<*eXtension*>> *(*p. *199)* Configures threads used by RTI Connext to receive and process data from transports (for example, UDP sockets).

Classes

- struct **DDS_ReceiverPoolQosPolicy**

  Configures threads used by RTI Connext to receive and process data from transports (for example, UDP sockets).

Variables

- const char *const **DDS_RECEIVERPOOL_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_ReceiverPoolQosPolicy*** *(*p. *862).*

5.127.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Configures threads used by RTI Connext to receive and process data from transports (for example, UDP sockets).

5.127.2 Variable Documentation

5.127.2.1 const char* const DDS_RECEIVERPOOL_QOS_POLICY_NAME

Stringified human-readable name for **DDS_ReceiverPoolQosPolicy** (p. 862).

5.128 PUBLISH_MODE

<<***eXtension***>> *(p. 199)* Specifies how RTI Connext sends application data on the network. This QoS policy can be used to tell RTI Connext to use its *own* thread to send data, instead of the user thread.

Classes

- struct **DDS_PublishModeQosPolicy**

  Specifies how RTI Connext sends application data on the network. This QoS policy can be used to tell RTI Connext to use its own *thread to send data, instead of the user thread.*

Defines

- #define **DDS_PUBLICATION_PRIORITY_UNDEFINED**

  Initializer value for ***DDS_PublishModeQosPolicy::priority*** *(p. 855) and/or* ***DDS_ChannelSettings_t::priority*** *(p. 487).*

- #define **DDS_PUBLICATION_PRIORITY_AUTOMATIC**

  Constant value for ***DDS_PublishModeQosPolicy::priority*** *(p. 855) and/or* ***DDS_ChannelSettings_t::priority*** *(p. 487).*

Enumerations

- enum **DDS_PublishModeQosPolicyKind** {
  DDS_SYNCHRONOUS_PUBLISH_MODE_QOS,
  DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS }

  Kinds of publishing mode.

Variables

- const char *const **DDS_PUBLISHMODE_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_PublishModeQosPolicy*** *(p. 853).*

5.128.1 Detailed Description

<<*eXtension*>> *(p. 199)* Specifies how RTI Connext sends application data on the network. This QoS policy can be used to tell RTI Connext to use its *own* thread to send data, instead of the user thread.

5.128.2 Define Documentation

5.128.2.1 #define DDS_PUBLICATION_PRIORITY_UNDEFINED

Initializer value for **DDS_PublishModeQosPolicy::priority** (p. 855) and/or **DDS_ChannelSettings_t::priority** (p. 487).

When assigned this value, the publication priority of the data writer, or channel of a multi-channel data writer, will be set to the lowest possible value. For muilti-channel data writers, if either the data writer or channel priority is NOT set to this value, then the publication priority of the entity will be set to the defined value.

5.128.2.2 #define DDS_PUBLICATION_PRIORITY_AUTOMATIC

Constant value for **DDS_PublishModeQosPolicy::priority** (p. 855) and/or **DDS_ChannelSettings_t::priority** (p. 487).

When assigned this value, the publication priority of the data writer, or channel of a multi-channel data writer, will be set to the largest priority value of any sample currently queued for publication by the data writer or data writer channel.

5.128.3 Enumeration Type Documentation

5.128.3.1 enum DDS_PublishModeQosPolicyKind

Kinds of publishing mode.

QoS:

DDS_PublishModeQosPolicy (p. 853)

Enumerator:

DDS_SYNCHRONOUS_PUBLISH_MODE_QOS Indicates to send data synchronously.

If **DDS_DataWriterProtocolQosPolicy::push_on_write** (p. 537) is **DDS_BOOLEAN_TRUE** (p. 298), data is sent immediately in the context of **FooDataWriter::write** (p. 1484).

As data is sent immediately in the context of the user thread, no flow control is applied.

See also:

DDS_DataWriterProtocolQosPolicy::push_on_write (p. 537)

[**default**] for **DDSDataWriter** (p. 1113)

DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS Indicates to send data asynchronously.

Configures the **DDSDataWriter** (p. 1113) to delegate the task of data transmission to a separate publishing thread. The **FooDataWriter::write** (p. 1484) call does not send the data, but instead schedules the data to be sent later by its associated **DDSPublisher** (p. 1346).

Each **DDSPublisher** (p. 1346) uses its dedicated publishing thread (**DDS_PublisherQos::asynchronous_publisher** (p. 852)) to send data for all its asynchronous DataWriters. For each asynchronous DataWriter, the associated **DDSFlowController** (p. 1259) determines when the publishing thread is allowed to send the data.

DDSDataWriter::wait_for_asynchronous_publishing (p. 1126) and **DDSPublisher::wait_for_asynchronous_publishing** (p. 1365) enable you to determine when the data has actually been sent.

See also:

DDSFlowController (p. 1259)
DDS_HistoryQosPolicy (p. 758)
DDSDataWriter::wait_for_asynchronous_publishing (p. 1126)
DDSPublisher::wait_for_asynchronous_publishing (p. 1365)
NDDS_Transport_Property_t::gather_send_buffer_count_max (p. 1525)

5.128.4 Variable Documentation

5.128.4.1 const char* const DDS_PUBLISHMODE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_PublishModeQosPolicy** (p. 853).

5.129 DISCOVERY_CONFIG

<<*eXtension*>> *(*p. 199*)* Specifies the discovery configuration QoS.

Classes

- struct **DDS_BuiltinTopicReaderResourceLimits_t**

  Built-in topic reader's resource limits.

- struct **DDS_DiscoveryConfigQosPolicy**

  Settings for discovery configuration.

Defines

- #define **DDS_DISCOVERYCONFIG_BUILTIN_PLUGIN_MASK_ALL**

  All bits are set.

- #define **DDS_DISCOVERYCONFIG_BUILTIN_PLUGIN_MASK_NONE**

  No bits are set.

- #define **DDS_DISCOVERYCONFIG_BUILTIN_PLUGIN_MASK_DEFAULT** (**DDS_DiscoveryConfigBuiltinPluginKindMask**)DDS_DISCOVERYCONFIG_BUILTIN_SDP

  The default value of ***DDS_DiscoveryConfigQosPolicy::builtin_discovery_plugins*** *(*p. 580*).*

Typedefs

- typedef **DDS_Long DDS_DiscoveryConfigBuiltinPluginKindMask**

  A bit-mask (list) of built-in discovery plugins.

Enumerations

- enum **DDS_DiscoveryConfigBuiltinPluginKind** { , **DDS_DISCOVERYCONFIG_BUILTIN_SDP** }

Built-in discovery plugins that can be used.

ˆ enum **DDS_RemoteParticipantPurgeKind** {

DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE,

DDS_NO_REMOTE_PARTICIPANT_PURGE }

Available behaviors for halting communication with remote participants (and their contained entities) with which discovery communication has been lost.

Variables

ˆ const char ∗const **DDS_DISCOVERYCONFIG_QOS_POLICY_NAME**

*Stringified human-readable name for **DDS_DiscoveryConfigQosPolicy*** *(p. 573)*.

5.129.1 Detailed Description

<<***eXtension***>> *(p. 199)* Specifies the discovery configuration QoS.

5.129.2 Define Documentation

5.129.2.1 #define DDS_DISCOVERYCONFIG_BUILTIN_PLUGIN_MASK_ALL

All bits are set.

See also:

DDS_DiscoveryConfigBuiltinPluginKindMask (p. 425)

5.129.2.2 #define DDS_DISCOVERYCONFIG_BUILTIN_PLUGIN_MASK_NONE

No bits are set.

See also:

DDS_DiscoveryConfigBuiltinPluginKindMask (p. 425)

5.129.2.3 #define DDS_DISCOVERYCONFIG_-BUILTIN_PLUGIN_MASK_DEFAULT (DDS_-DiscoveryConfigBuiltinPluginKindMask)DDS_-DISCOVERYCONFIG_BUILTIN_SDP

The default value of **DDS_DiscoveryConfigQosPolicy::builtin_-discovery_plugins** (p. 580).

See also:

DDS_DiscoveryConfigBuiltinPluginKindMask (p. 425)

5.129.3 Typedef Documentation

5.129.3.1 typedef DDS_Long DDS_-DiscoveryConfigBuiltinPluginKindMask

A bit-mask (list) of built-in discovery plugins.

The bit-mask is an efficient and compact representation of a fixed-length list of **DDS_DiscoveryConfigBuiltinPluginKind** (p. 425) values.

QoS:

DDS_DiscoveryConfigQosPolicy (p. 573)

5.129.4 Enumeration Type Documentation

5.129.4.1 enum DDS_DiscoveryConfigBuiltinPluginKind

Built-in discovery plugins that can be used.

See also:

DDS_DiscoveryConfigBuiltinPluginKindMask (p. 425)

Enumerator:

DDS_DISCOVERYCONFIG_BUILTIN_SDP **[default]** Simple discovery plugin.

5.129.4.2 enum DDS_RemoteParticipantPurgeKind

Available behaviors for halting communication with remote participants (and their contained entities) with which discovery communication has been lost.

When discovery communication with a remote participant has been lost, the local participant must make a decision about whether to continue attempting to communicate with that participant and its contained entities. This "kind" is used to select the desired behavior.

This "kind" does not pertain to the situation in which a remote participant has been gracefully deleted and notification of that deletion have been successfully received by its peers. In that case, the local participant will immediately stop attempting to communicate with those entities and will remove the associated remote entity records from its internal database.

See also:

DDS_DiscoveryConfigQosPolicy::remote_participant_purge_kind (p. 576)

Enumerator:

DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE [**default**] Maintain knowledge of the remote participant for as long as it maintains its liveliness contract.

A participant will continue attempting communication with its peers, even if discovery communication with them is lost, as long as the remote participants maintain their liveliness. If both discovery communication and participant liveliness are lost, however, the local participant will remove all records of the remote participant and its contained endpoints, and no further data communication with them will occur until and unless they are rediscovered.

The liveliness contract a participant promises to its peers – its "liveliness lease duration" – is specified in its **DDS_DiscoveryConfigQosPolicy::participant_liveliness_lease_duration** (p. 575) QoS field. It maintains that contract by writing data to those other participants with a writer that has a **DDS_LivelinessQosPolicyKind** (p. 358) of **DDS_AUTOMATIC_LIVELINESS_QOS** (p. 359) or **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** (p. 359) and by asserting itself (at the **DDS_DiscoveryConfigQosPolicy::participant_liveliness_assert_period** (p. 576)) over the Simple Discovery Protocol.

DDS_NO_REMOTE_PARTICIPANT_PURGE Never "forget" a remote participant with which discovery communication has been lost.

If a participant with this behavior loses discovery communication with a remote participant, it will nevertheless remember that remote participant and its endpoints and continue attempting to communicate with them indefinitely.

This value has consequences for a participant's resource usage. If discovery communication with a remote participant is lost, but the

same participant is later rediscovered, any relevant records that remain in the database will be reused. However, if it is not rediscovered, the records will continue to take up space in the database for as long as the local participant remains in existence.

5.129.5 Variable Documentation

5.129.5.1 const char* const DDS_DISCOVERYCONFIG_QOS_POLICY_NAME

Stringified human-readable name for **DDS_DiscoveryConfigQosPolicy** (p. 573).

5.130 ASYNCHRONOUS_PUBLISHER

<<*eXtension*>> *(p. 199)* Specifies the asynchronous publishing settings of the **DDSPublisher** (p. 1346) instances.

Classes

- struct **DDS_AsynchronousPublisherQosPolicy**

  Configures the mechanism that sends user data in an external middleware thread.

Variables

- const char *const **DDS_ASYNCHRONOUSPUBLISHER_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_AsynchronousPublisherQosPolicy** (p. 466).*

5.130.1 Detailed Description

<<*eXtension*>> *(p. 199)* Specifies the asynchronous publishing settings of the **DDSPublisher** (p. 1346) instances.

5.130.2 Variable Documentation

5.130.2.1 const char* const DDS_ASYNCHRONOUSPUBLISHER_QOS_POLICY_NAME

Stringified human-readable name for **DDS_AsynchronousPublisherQosPolicy** (p. 466).

5.131 TYPESUPPORT

<<***eXtension***>> *(*p. *199**)* Allows you to attach application-specific values to a DataWriter or DataReader that are passed to the serialization or deserialization routine of the associated data type.

Classes

- struct **DDS_TypeSupportQosPolicy**

  Allows you to attach application-specific values to a DataWriter or DataReader that are passed to the serialization or deserialization routine of the associated data type.

Variables

- const char *const **DDS_TYPESUPPORT_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_TypeSupportQosPolicy*** *(*p. *1040**).*

5.131.1 Detailed Description

<<***eXtension***>> *(*p. *199**)* Allows you to attach application-specific values to a DataWriter or DataReader that are passed to the serialization or deserialization routine of the associated data type.

5.131.2 Variable Documentation

5.131.2.1 const char* const DDS_TYPESUPPORT_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TypeSupportQosPolicy** (p. 1040).

5.132 EXCLUSIVE_AREA

*<<**eXtension**>>* *(p. 199)* Configures multi-thread concurrency and deadlock prevention capabilities.

Classes

- struct **DDS_ExclusiveAreaQosPolicy**

  Configures multi-thread concurrency and deadlock prevention capabilities.

Variables

- const char *const **DDS_EXCLUSIVEAREA_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_ExclusiveAreaQosPolicy** (p. 742).*

5.132.1 Detailed Description

*<<**eXtension**>>* *(p. 199)* Configures multi-thread concurrency and deadlock prevention capabilities.

5.132.2 Variable Documentation

5.132.2.1 const char* const DDS_EXCLUSIVEAREA_QOS_POLICY_NAME

Stringified human-readable name for **DDS_ExclusiveAreaQosPolicy** (p. 742).

5.133 BATCH

<<*eXtension*>> *(*p. *199)* Batch QoS policy used to enable batching in **DDSDataWriter** (p. 1113) instances.

Classes

- struct **DDS_BatchQosPolicy**

  Used to configure batching of multiple samples into a single network packet in order to increase throughput for small samples.

Variables

- const char *const **DDS_BATCH_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_BatchQosPolicy*** *(*p. *476).*

5.133.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Batch QoS policy used to enable batching in **DDSDataWriter** (p. 1113) instances.

5.133.2 Variable Documentation

5.133.2.1 const char* const DDS_BATCH_QOS_POLICY_NAME

Stringified human-readable name for **DDS_BatchQosPolicy** (p. 476).

5.134 TYPE_CONSISTENCY_-ENFORCEMENT

Defines the rules for determining whether the type used to publish a given topic is consistent with that used to subscribe to it.

Classes

- struct **DDS_TypeConsistencyEnforcementQosPolicy**

  Defines the rules for determining whether the type used to publish a given topic is consistent with that used to subscribe to it.

Enumerations

- enum **DDS_TypeConsistencyKind** {
  DDS_DISALLOW_TYPE_COERCION,
  DDS_ALLOW_TYPE_COERCION }

  Kinds of type consistency.

Variables

- const char *const **DDS_TYPE_CONSISTENCY_-ENFORCEMENT_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_-TypeConsistencyEnforcementQosPolicy*** (p. 1038).

5.134.1 Detailed Description

Defines the rules for determining whether the type used to publish a given topic is consistent with that used to subscribe to it.

5.134.2 Enumeration Type Documentation

5.134.2.1 enum DDS_TypeConsistencyKind

Kinds of type consistency.

QoS:

DDS_TypeConsistencyEnforcementQosPolicy (p. 1038)

Enumerator:

DDS_DISALLOW_TYPE_COERCION The DataWriter and the DataReader must support the same data type in order for them to communicate.

This is the degree of type consistency enforcement required by the `OMG DDS Specification` prior to the `OMG Extensible and Dynamic Topic Types for DDS Specification`.

DDS_ALLOW_TYPE_COERCION The DataWriter and the DataReader need not support the same data type in order for them to communicate as long as the DataReaders type is assignable from the DataWriters type.

For example, the following two extensible types will be assignable to each other since MyDerivedType contains all the members of MyBaseType (member_1) plus some additional elements (member_2).

```
struct MyBaseType {
    long member_1;
};

struct MyDerivedType: MyBaseType {
    long member_2;
};
```

Even if MyDerivedType was not explicitly inheriting from MyBaseType the types would still be assignable. For example:

```
struct MyBaseType {
    long member_1;
};

struct MyDerivedType {
    long member_1;
    long member_2;
};
```

For additional information on type assignability refer to the `OMG Extensible and Dynamic Topic Types for DDS Specification`. [**default**]

5.134.3 Variable Documentation

5.134.3.1 const char* const DDS_TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_NAME

Stringified human-readable name for **DDS_TypeConsistencyEnforcementQosPolicy** (p. 1038).

5.135 LOCATORFILTER

<<*eXtension*>> *(p. 199)* The QoS policy used to report the configuration of a MultiChannel DataWriter as part of **DDS_PublicationBuiltinTopicData** (p. 839).

Classes

- struct **DDS_LocatorFilter_t**

  Specifies the configuration of an individual channel within a MultiChannel DataWriter.

- struct **DDS_LocatorFilterSeq**

  Declares IDL `sequence<` ***DDS_LocatorFilter_t*** *(p. 785)* `>`.

- struct **DDS_LocatorFilterQosPolicy**

  The QoS policy used to report the configuration of a MultiChannel DataWriter as part of ***DDS_PublicationBuiltinTopicData*** *(p. 839).*

Variables

- const char *const **DDS_LOCATORFILTER_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_LocatorFilterQosPolicy*** *(p. 787).*

5.135.1 Detailed Description

<<*eXtension*>> *(p. 199)* The QoS policy used to report the configuration of a MultiChannel DataWriter as part of **DDS_PublicationBuiltinTopicData** (p. 839).

5.135.2 Variable Documentation

5.135.2.1 const char* const DDS_LOCATORFILTER_QOS_POLICY_NAME

Stringified human-readable name for **DDS_LocatorFilterQosPolicy** (p. 787).

5.136 MULTICHANNEL

<<***eXtension***>> *(*p. *199**)* Configures the ability of a DataWriter to send data on different multicast groups (addresses) based on the value of the data.

Classes

- struct **DDS_ChannelSettings_t**

  Type used to configure the properties of a channel.

- struct **DDS_ChannelSettingsSeq**

  Declares IDL `sequence<` ***DDS_ChannelSettings_t*** *(*p. *486**)* `>`.

- struct **DDS_MultiChannelQosPolicy**

  Configures the ability of a DataWriter to send data on different multicast groups (addresses) based on the value of the data.

Variables

- const char *const **DDS_MULTICHANNEL_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_MultiChannelQosPolicy*** *(*p. *796**).*

5.136.1 Detailed Description

<<***eXtension***>> *(*p. *199**)* Configures the ability of a DataWriter to send data on different multicast groups (addresses) based on the value of the data.

5.136.2 Variable Documentation

5.136.2.1 const char* const DDS_MULTICHANNEL_QOS_POLICY_NAME

Stringified human-readable name for **DDS_MultiChannelQosPolicy** (p. 796).

5.137 PROPERTY

<<***eXtension***>> *(*p. *199)* Stores name/value (string) pairs that can be used to configure certain parameters of RTI Connext that are not exposed through formal QoS policies. Can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery.

Classes

- class **DDSPropertyQosPolicyHelper**

  Policy Helpers which facilitate management of the properties in the input policy.

- struct **DDS_Property_t**

  Properties are name/value pairs objects.

- struct **DDS_PropertySeq**

  Declares IDL `sequence` *<* ***DDS_Property_t*** *(* p. *833)* *>.*

- struct **DDS_PropertyQosPolicy**

  Stores name/value(string) pairs that can be used to configure certain parameters of RTI Connext that are not exposed through formal QoS policies. Can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery.

Functions

- static **DDS_Long DDSPropertyQosPolicyHelper::get_number_of_properties** (**DDS_PropertyQosPolicy** &policy)

  Gets the number of properties in the input policy.

- static **DDS_ReturnCode_t DDSPropertyQosPolicyHelper::assert_property** (**DDS_PropertyQosPolicy** &policy, const char *name, const char *value, **DDS_Boolean** propagate)

  Asserts the property identified by name in the input policy.

- static **DDS_ReturnCode_t DDSPropertyQosPolicyHelper::add_property** (**DDS_PropertyQosPolicy** &policy, const char *name, const char *value, **DDS_Boolean** propagate)

  Adds a new property to the input policy.

- static struct **DDS_Property_t** * **DDSPropertyQosPolicyHelper::lookup_property** (**DDS_PropertyQosPolicy** &policy, const char *name)

  Searches for a property in the input policy given its name.

- static **DDS_ReturnCode_t** **DDSPropertyQosPolicyHelper::remove_property** (**DDS_PropertyQosPolicy** &policy, const char *name)

  Removes a property from the input policy.

- static **DDS_ReturnCode_t** **DDSPropertyQosPolicyHelper::get_properties** (**DDS_PropertyQosPolicy** &policy, struct **DDS_PropertySeq** &properties, const char *name_prefix)

  Retrieves a list of properties whose names match the input prefix.

Variables

- const char *const **DDS_PROPERTY_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_PropertyQosPolicy** (p. 834).*

5.137.1 Detailed Description

<<***eXtension***>> *(p. 199)* Stores name/value (string) pairs that can be used to configure certain parameters of RTI Connext that are not exposed through formal QoS policies. Can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery.

RTI Connext will automatically set some system properties in the **DDS_PropertyQosPolicy** (p. 834) associated with a **DDS_DomainParticipantQos** (p. 588). See **System Properties** (p. 150) for additional details.

5.137.2 Function Documentation

5.137.2.1 static DDS_Long DDSPropertyQosPolicyHelper::get_number_of_properties (DDS_PropertyQosPolicy & *policy*) [static, inherited]

Gets the number of properties in the input policy.

Precondition:

policy cannot be NULL.

Parameters:

policy *<<in>>* *(*p. *200)* Input policy.

Returns:

Number of properties.

5.137.2.2 static DDS_ReturnCode_t DDSPropertyQosPolicyHelper::assert_property (DDS_PropertyQosPolicy & *policy*, const char * *name*, const char * *value*, DDS_Boolean *propagate*) [static, inherited]

Asserts the property identified by name in the input policy.

If the property already exists, this function replaces its current value with the new one.

If the property identified by name does not exist, this function adds it to the property set.

This function increases the maximum number of elements of the policy sequence when this number is not enough to store the new property.

Precondition:

policy, name and value cannot be NULL.

Parameters:

policy *<<in>>* *(*p. *200)* Input policy.

name *<<in>>* *(*p. *200)* Property name.

value *<<in>>* *(*p. *200)* Property value.

propagate *<<in>>* *(*p. *200)* Indicates if the property will be propagated on discovery.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

5.137.2.3 static DDS_ReturnCode_t DDSPropertyQosPolicyHelper::add_property (DDS_PropertyQosPolicy & *policy*, const char * *name*, const char * *value*, DDS_Boolean *propagate*) [static, inherited]

Adds a new property to the input policy.

This function will allocate memory to store the (name,value) pair. The memory allocated is owned by RTI Connext.

If the maximum number of elements of the policy sequence is not enough to store the new property, this function will increase it.

If the property already exists the function fails with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Precondition:

policy, name and value cannot be NULL.
The property is not in the policy.

Parameters:

policy <<*in*>> *(p. 200)* Input policy.

name <<*in*>> *(p. 200)* Property name.

value <<*in*>> *(p. 200)* Property value.

propagate <<*in*>> *(p. 200)* Indicates if the property will be propagated on discovery.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

5.137.2.4 static struct DDS_Property_t* DDSPropertyQosPolicyHelper::lookup_property (DDS_PropertyQosPolicy & *policy*, const char * *name*) [static, read, inherited]

Searches for a property in the input policy given its name.

Precondition:

policy, name and value cannot be NULL.

Parameters:

policy *<<in>>* *(p. 200)* Input policy.

name *<<in>>* *(p. 200)* Property name.

Returns:

On success, the function returns the first property with the given name. Otherwise, the function returns NULL.

5.137.2.5 static DDS_ReturnCode_t DDSPropertyQosPolicyHelper::remove_property (DDS_PropertyQosPolicy & *policy*, const char * *name*) [static, inherited]

Removes a property from the input policy.

If the property does not exist, the function fails with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Precondition:

policy and name cannot be NULL.
The property is in the policy.

Parameters:

policy *<<in>>* *(p. 200)* Input policy.

name *<<in>>* *(p. 200)* Property name.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

5.137.2.6 static DDS_ReturnCode_t DDSPropertyQosPolicyHelper::get_properties (DDS_PropertyQosPolicy & *policy*, struct DDS_PropertySeq & *properties*, const char * *name_prefix*) [static, inherited]

Retrieves a list of properties whose names match the input prefix.

If the properties sequence doesn't own its buffer, and its maximum is less than the total number of properties matching the input prefix, it will be filled up to its maximum and fail with an error of **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

Precondition:

policy, properties and name_prefix cannot be NULL.

Parameters:

policy *<<in>>* *(p. 200)* Input policy.

properties *<<inout>>* *(p. 200)* A **DDS_PropertySeq** (p. 837) object where the set or list of properties will be returned.

name_prefix Name prefix.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

5.137.3 Variable Documentation

5.137.3.1 const char* const DDS_PROPERTY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_PropertyQosPolicy** (p. 834).

5.138 AVAILABILITY

<<*eXtension*>> *(p. 199)* Configures the availability of data.

Classes

- struct **DDS_EndpointGroup_t**

  Specifies a group of endpoints that can be collectively identified by a name and satisfied by a quorum.

- struct **DDS_EndpointGroupSeq**

  A sequence of ***DDS_EndpointGroup_t*** *(p. 731).*

- struct **DDS_AvailabilityQosPolicy**

  Configures the availability of data.

Variables

- const char *const **DDS_AVAILABILITY_QOS_POLICY_NAME**

  Stringified human-readable name for ***DDS_AvailabilityQosPolicy*** *(p. 471).*

5.138.1 Detailed Description

<<*eXtension*>> *(p. 199)* Configures the availability of data.

5.138.2 Variable Documentation

5.138.2.1 const char* const DDS_AVAILABILITY_QOS_POLICY_NAME

Stringified human-readable name for **DDS_AvailabilityQosPolicy** (p. 471).

5.139 Entity Support

DDSEntity (p. 1253), **DDSListener** (p. 1318) and related items.

Classes

- class **DDSListener**

  <<**interface**>> (p. 199) *Abstract base class for all Listener interfaces.*

- class **DDSEntity**

  <<**interface**>> (p. 199) *Abstract base class for all the DDS objects that support QoS policies, a listener, and a status condition.*

- class **DDSDomainEntity**

  <<**interface**>> (p. 199) *Abstract base class for all DDS entities except for the **DDSDomainParticipant** (p. 1139).*

5.139.1 Detailed Description

DDSEntity (p. 1253), **DDSListener** (p. 1318) and related items.

DDSEntity (p. 1253) subtypes are created and destroyed by factory objects. With the exception of **DDSDomainParticipant** (p. 1139), whose factory is **DDSDomainParticipantFactory** (p. 1216), all **DDSEntity** (p. 1253) factory objects are themselves **DDSEntity** (p. 1253) subtypes as well.

Important: all **DDSEntity** (p. 1253) delete operations are inherently thread-unsafe. The user must take extreme care that a given **DDSEntity** (p. 1253) is not destroyed in one thread while being used concurrently (including being deleted concurrently) in another thread. An operation's effect in the presence of the concurrent deletion of the operation's target **DDSEntity** (p. 1253) is undefined.

5.140 Conditions and WaitSets

DDSCondition (p. 1075) and **DDSWaitSet** (p. 1433) and related items.

Classes

- struct **DDSConditionSeq**

  Instantiates `FooSeq` (p. 1494) < ***DDSCondition*** *(p. 1075)* >.

- class **DDSCondition**

  <<**interface**>> (p. 199) *Root class for all the conditions that may be attached to a* ***DDSWaitSet*** *(p. 1433).*

- class **DDSGuardCondition**

  <<**interface**>> (p. 199) *A specific* ***DDSCondition*** *(p. 1075) whose* `trigger_value` *is completely under the control of the application.*

- class **DDSStatusCondition**

  <<**interface**>> (p. 199) *A specific* ***DDSCondition*** *(p. 1075) that is associated with each* ***DDSEntity*** *(p. 1253).*

- class **DDSWaitSet**

  <<**interface**>> (p. 199) *Allows an application to wait until one or more of the attached* ***DDSCondition*** *(p. 1075) objects has a* `trigger_value` *of* ***DDS_BOOLEAN_TRUE*** *(p. 298) or else until the timeout expires.*

- struct **DDS_WaitSetProperty_t**

  <<**eXtension**>> (p. 199) *Specifies the* ***DDSWaitSet*** *(p. 1433) behavior for multiple trigger events.*

5.140.1 Detailed Description

DDSCondition (p. 1075) and **DDSWaitSet** (p. 1433) and related items.

5.141 ENTITY_NAME

<<*eXtension*>> *(p. 199)* Assigns a name to a **DDSDomainParticipant** (p. 1139). This name will be visible during the discovery process and in RTI tools to help you visualize and debug your system.

Classes

- struct **DDS_EntityNameQosPolicy**

  *Assigns a name and a role name to a **DDSDomainParticipant** (p. 1139), **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087). These names will be visible during the discovery process and in RTI tools to help you visualize and debug your system.*

Variables

- const char *const **DDS_ENTITYNAME_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_EntityNameQosPolicy** (p. 735).*

5.141.1 Detailed Description

<<*eXtension*>> *(p. 199)* Assigns a name to a **DDSDomainParticipant** (p. 1139). This name will be visible during the discovery process and in RTI tools to help you visualize and debug your system.

5.141.2 Variable Documentation

5.141.2.1 const char* const DDS_ENTITYNAME_QOS_POLICY_NAME

Stringified human-readable name for **DDS_EntityNameQosPolicy** (p. 735).

5.142 PROFILE

<<*eXtension*>> *(p. 199)* Configures the way that XML documents containing QoS profiles are loaded by RTI Connext.

Classes

- struct **DDS_ProfileQosPolicy**

  Configures the way that XML documents containing QoS profiles are loaded by RTI Connext.

Variables

- const char *const **DDS_PROFILE_QOS_POLICY_NAME**

  *Stringified human-readable name for **DDS_ProfileQosPolicy** (p. 830).*

5.142.1 Detailed Description

<<*eXtension*>> *(p. 199)* Configures the way that XML documents containing QoS profiles are loaded by RTI Connext.

5.142.2 Variable Documentation

5.142.2.1 const char* const DDS_PROFILE_QOS_POLICY_NAME

Stringified human-readable name for **DDS_ProfileQosPolicy** (p. 830).

5.143 WriteParams

*<<**eXtension**>> (p. 199)*

Classes

- struct **DDS_SampleIdentity_t**

  Type definition for an Sample Identity.

- struct **DDS_AckResponseData_t**

  Data payload of an application-level acknowledgment.

- struct **DDS_WriteParams_t**

  <<**eXtension**>> (p. 199) *Input parameters for writing with **FooDataWriter::write_w_params** (p. 1487), **FooDataWriter::dispose_w_params** (p. 1491), **FooDataWriter::register_instance_w_params** (p. 1480), **FooDataWriter::unregister_instance_w_params** (p. 1483)*

Functions

- **DDS_Boolean DDS_SampleIdentity_equals** (const struct **DDS_SampleIdentity_t** *self, const struct **DDS_SampleIdentity_t** *other)

  Compares this sample identity with another sample identity for equality.

- void **DDS_WriteParams_reset** (struct **DDS_WriteParams_t** *self)

  Resets all the fields to their default values.

Variables

- struct **DDS_GUID_t DDS_SampleIdentity_t::writer_guid**

  16-byte identifier identifying the virtual GUID.

- struct **DDS_SequenceNumber_t DDS_SampleIdentity_t::sequence_number**

  monotonically increasing 64-bit integer that identifies the sample in the data source.

- struct **DDS_SampleIdentity_t DDS_AUTO_SAMPLE_IDENTITY**

The AUTO sample identity.

ˆ struct **DDS_SampleIdentity_t DDS_UNKNOWN_SAMPLE_IDENTITY**

An invalid or unknown sample identity.

ˆ struct **DDS_WriteParams_t DDS_WRITEPARAMS_DEFAULT**

*Initializer for **DDS_WriteParams_t** (p. 1067).*

5.143.1 Detailed Description

*<<**eXtension**>> (p. 199)*

5.143.2 Function Documentation

5.143.2.1 DDS_Boolean DDS_SampleIdentity_equals (const struct DDS_SampleIdentity_t * *self*, const struct DDS_SampleIdentity_t * *other*)

Compares this sample identity with another sample identity for equality.

Parameters:

self *<<**in**>> (p. 200)* This sample identity.

other *<<**in**>> (p. 200)* The other sample identity to be compared with this sample identity.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the two sample identities have equal values, or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

5.143.2.2 void DDS_WriteParams_reset (struct DDS_WriteParams_t * *self*)

Resets all the fields to their default values.

This operation is useful to reset all the fields to their automatic value when **DDS_WriteParams_t::replace_auto** (p. 1067) is enabled and the same params instance is used in multiple calls to **FooDataWriter::write_w_params** (p. 1487)

5.143.3 Variable Documentation

5.143.3.1 struct DDS_GUID_t DDS_SampleIdentity_t::writer_guid [read, inherited]

16-byte identifier identifying the virtual GUID.

5.143.3.2 struct DDS_SequenceNumber_t DDS_SampleIdentity_t::sequence_number [read, inherited]

monotonically increasing 64-bit integer that identifies the sample in the data source.

5.143.3.3 struct DDS_SampleIdentity_t DDS_AUTO_SAMPLE_IDENTITY

The AUTO sample identity.

Special **DDS_AUTO_SAMPLE_IDENTITY** (p. 449) value {**DDS_GUID_AUTO** (p. 309), **DDS_AUTO_SEQUENCE_NUMBER** (p. 311)}

5.143.3.4 struct DDS_SampleIdentity_t DDS_UNKNOWN_SAMPLE_IDENTITY

An invalid or unknown sample identity.

Special **DDS_UNKNOWN_SAMPLE_IDENTITY** (p. 449) value {**DDS_GUID_UNKNOWN** (p. 309), **DDS_SEQUENCE_NUMBER_UNKNOWN** (p. 310)}

5.143.3.5 struct DDS_WriteParams_t DDS_WRITEPARAMS_DEFAULT

Initializer for **DDS_WriteParams_t** (p. 1067).

5.144 LOGGING

<<*eXtension*>> *(*p. *199)* Configures the RTI Connext logging facility.

Classes

ˆ struct **DDS_LoggingQosPolicy**

Configures the RTI Connext logging facility.

Variables

ˆ const char ∗const **DDS_LOGGING_QOS_POLICY_NAME**

Stringified human-readable name for ***DDS_LoggingQosPolicy*** *(*p. *791).*

5.144.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Configures the RTI Connext logging facility.

5.144.2 Variable Documentation

5.144.2.1 const char∗ const DDS_LOGGING_QOS_POLICY_NAME

Stringified human-readable name for **DDS_LoggingQosPolicy** (p. 791).

5.145 Octet Buffer Support

<<*eXtension*>> *(*p. *199)* Octet buffer creation, cloning, and deletion.

Functions

- unsigned char * **DDS_OctetBuffer_alloc** (unsigned int size)

  Create a new empty OctetBuffer that can hold up to `size` *octets.*

- unsigned char * **DDS_OctetBuffer_dup** (const unsigned char *buffer, unsigned int size)

  Clone an OctetBuffer.

- void **DDS_OctetBuffer_free** (unsigned char *buffer)

  Delete an OctetBuffer.

5.145.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Octet buffer creation, cloning, and deletion.

The methods in this class ensure consistent cross-platform implementations for OctetBuffer creation (**DDS_OctetBuffer_alloc()** (p. 453)), deletion (**DDS_OctetBuffer_free()** (p. 453)), and cloning (**DDS_OctetBuffer_dup()** (p. 453)) that preserve the mutable value type semantics. These are to be viewed as methods that define an `OctetBuffer` class whose data is represented by a '`unsigned` char*'.

5.145.2 Conventions

The following conventions govern the memory management of OctetBuffers in RTI Connext.

- The DDS implementation ensures that when value types containing OctetBuffers are passed back and forth to the DDS APIs, the OctetBuffers are created/deleted/cloned using the `OctetBuffer` class methods.
  - Value types containing OctetBuffers have ownership of the contained OctetBuffer. Thus, when a value type is deleted, the contained octet buffer field is also deleted.
- The user must ensure that when value types containing OctetBuffers are passed back and forth to the DDS APIs, the OctetBuffers are created/deleted/cloned using the OctetBuffer class methods.

The representation of an OctetBuffer in C/C++ unfortunately does not allow programs to detect how much memory has been allocated for a OctetBuffer. RTI Connext must therefore make some assumptions when a user requests that a OctetBuffer be copied into. The following rules apply when RTI Connext is copying into an OctetBuffer.

- If the 'unsigned char*' is NULL, RTI Connext will allocate a new OctetBuffer on behalf of the user. *To avoid leaking memory, you must ensure that the OctetBuffer will be freed (see* ***Usage*** *(p. 452) below)* *in C. For C++, the destructor of the valuetype containing the OctetBuffer will free it automatically.*.

- If the 'unsigned char*' is not NULL, RTI Connext will assume that you are managing the OctetBuffer's memory yourself and have allocated enough memory to store the OctetBuffer to be copied. *RTI Connext will copy into your memory; to avoid memory corruption, be sure to allocate enough of it.* Also, *do not pass structures containing junk pointers into RTI Connext*; you are likely to crash.

5.145.3 Usage

This requirement can generally be assured by adhering to the following *idiom* for manipulating OctetBuffers.

```
Always use
     DDS_OctetBuffer_alloc() to create,
     DDS_OctetBuffer_dup() to clone,
     DDS_OctetBuffer_free() to delete
 a 'unsigned char*' that is passed back and forth between
 user code and the DDS C/C++ APIs.
```

Not adhering to this idiom can result in bad pointers, and incorrect memory being freed.

In addition, the user code should be vigilant to avoid memory leaks. It is good practice to:

- Balance occurrences of **DDS_OctetBuffer_alloc()** (p. 453), with matching occurrences of **DDS_OctetBuffer_free()** (p. 453) in the code.

- Finalize value types containing OctetBuffer. In C++ the destructor accomplishes this automatically. in C, explicit "destructor" functions are provided.

5.145.4 Function Documentation

5.145.4.1 unsigned char* DDS_OctetBuffer_alloc (unsigned int *size*)

Create a new empty OctetBuffer that can hold up to `size` octets.

An OctetBuffer created by this method must be deleted using **DDS_OctetBuffer_free()** (p. 453).

This function will allocate enough memory to hold an OctetBuffer of `size` octets.

Parameters:

size *<<in>>* *(p. 200)* Size of the buffer.

Returns:

A newly created non-NULL OctetBuffer upon success or NULL upon failure.

5.145.4.2 unsigned char* DDS_OctetBuffer_dup (const unsigned char * *buffer*, unsigned int *size*)

Clone an OctetBuffer.

An OctetBuffer created by this method must be deleted using **DDS_OctetBuffer_free()** (p. 453)

Parameters:

buffer *<<in>>* *(p. 200)* The OctetBuffer to duplicate.

size *<<in>>* *(p. 200)* Size of the OctetBuffer to duplicate.

Returns:

If `src` == NULL or size <0, this method always returns NULL. Otherwise, upon success it returns a newly created OctetBuffer whose value is `src`; upon failure it returns NULL.

5.145.4.3 void DDS_OctetBuffer_free (unsigned char * *buffer*)

Delete an OctetBuffer.

Precondition:

`buffer` must be either NULL, or must have been created using **DDS_OctetBuffer_alloc()** (p. 453), **DDS_OctetBuffer_dup()** (p. 453)

Parameters:

buffer <<*in*>> *(*p. *200)* The buffer to delete.

5.146 Sequence Support

The **FooSeq** (p. 1494) interface allows you to work with variable-length collections of homogeneous data.

Modules

- **Built-in Sequences**

  Defines sequences of primitive data type.

Classes

- struct **FooSeq**

  <<**interface**>> (p. 199) <<**generic**>> (p. 199) *A type-safe, ordered collection of elements. The type of these elements is referred to in this documentation as* `Foo` (p. 1443).

5.146.1 Detailed Description

The **FooSeq** (p. 1494) interface allows you to work with variable-length collections of homogeneous data.

This interface is instantiated for each concrete element type in order to provide compile-time type safety to applications. The **Built-in Sequences** (p. 120) are pre-defined instantiations for the primitive data types.

When you use the **rtiddsgen** (p. 220) code generation tool, it will automatically generate concrete sequence instantiations for each of your own custom types.

5.147 String Support

<<*eXtension*>> *(*p. *199)* String creation, cloning, assignment, and deletion.

Functions

- char * **DDS_String_alloc** (size_t length)

  Create a new empty string that can hold up to `length` *characters.*

- char * **DDS_String_dup** (const char *str)

  Clone a string. Creates a new string that duplicates the value of `string`*.*

- void **DDS_String_free** (char *str)

  Delete a string.

- **DDS_Wchar** * **DDS_Wstring_alloc** (**DDS_UnsignedLong** length)

  Create a new empty string that can hold up to `length` *wide characters.*

- **DDS_UnsignedLong** **DDS_Wstring_length** (const **DDS_Wchar** *str)

  Get the number of wide characters in the given string.

- **DDS_Wchar** * **DDS_Wstring_copy** (**DDS_Wchar** *dst, const **DDS_Wchar** *src)

  Copy the source string over the destination string reallocating the space if it's necessary.

- **DDS_Wchar** * **DDS_Wstring_copy_and_widen** (**DDS_Wchar** *dst, const char *src)

  Copy the source string over the destination string, widening each character.

- **DDS_Wchar** * **DDS_Wstring_dup** (const **DDS_Wchar** *str)

  Clone a string of wide characters. Creates a new string that duplicates the value of `string`*.*

- **DDS_Wchar** * **DDS_Wstring_dup_and_widen** (const char *str)

  Clone a string of characters as a string of wide characters.

- void **DDS_Wstring_free** (**DDS_Wchar** *str)

  Delete a string.

5.147.1 Detailed Description

<<*eXtension*>> *(p. 199)* String creation, cloning, assignment, and deletion.

The methods in this class ensure consistent cross-platform implementations for string creation (**DDS_String_alloc()** (p. 458)), deletion (**DDS_String_free()** (p. 459)), and cloning (**DDS_String_dup()** (p. 459)) that preserve the mutable value type semantics. These are to be viewed as methods that define a `string` class whose data is represented by a `'char*'`.

5.147.2 Conventions

The following conventions govern the memory management of strings in RTI Connext.

- The DDS implementation ensures that when value types containing strings are passed back and forth to the DDS APIs, the strings are created/deleted/assigned/cloned using the `string` class methods.
  - Value types containing strings have ownership of the contained string. Thus, when a value type is deleted, the contained string field is also deleted.
  - **DDS_StringSeq** (p. 929) is a value type that contains strings; it owns the memory for the contained strings. When a **DDS_StringSeq** (p. 929) is assigned or deleted, the contained strings are also assigned or deleted respectively.
- The user must ensure that when value types containing strings are passed back and forth to the DDS APIs, the strings are created/deleted/assigned/cloned using the String class methods.

The representation of a string in C/C++ unfortunately does not allow programs to detect how much memory has been allocated for a string. RTI Connext must therefore make some assumptions when a user requests that a string be copied into. The following rules apply when RTI Connext is copying into a string or string sequence:

- If the 'char*' is NULL, RTI Connext will log a warning and allocate a new string on behalf of the user. *To avoid leaking memory, you must ensure that the string will be freed (see **Usage** (p. 458) below).*
- If the 'char*' is not NULL, RTI Connext will assume that you are managing the string's memory yourself and have allocated enough memory to store the string to be copied. *RTI Connext will copy into your memory; to avoid memory corruption, be sure to allocate enough of it.* Also, *do not pass structures containing junk pointers into RTI Connext*; you are likely to crash.

5.147.3 Usage

This requirement can generally be assured by adhering to the following *idiom* for manipulating strings.

```
Always use
     DDS_String_alloc() to create,
     DDS_String_dup() to clone,
     DDS_String_free() to delete
 a string 'char*' that is passed back and forth between
 user code and the DDS C/C++ APIs.
```

Not adhering to this idiom can result in bad pointers, and incorrect memory being freed.

In addition, the user code should be vigilant to avoid memory leaks. It is good practice to:

- Balance occurrences of **DDS_String_alloc()** (p. 458), **DDS_String_dup()** (p. 459), with matching occurrences of **DDS_String_free()** (p. 459) in the code.
- Finalize value types containing strings. In C++ the destructor accomplishes this automatically. in C, explicit "destructor" functions are provided; these functions are typically called "finalize."

See also:

DDS_StringSeq (p. 929)

5.147.4 Function Documentation

5.147.4.1 char* DDS_String_alloc (size_t *length*)

Create a new empty string that can hold up to `length` characters.

A string created by this method must be deleted using **DDS_String_free()** (p. 459).

This function will allocate enough memory to hold a string of `length` characters, **plus** one additional byte to hold the NULL terminating character.

Parameters:

length *<<in>> (p. 200)* Capacity of the string.

Returns:

A newly created non-NULL string upon success or NULL upon failure.

Examples:

HelloWorld.cxx.

5.147.4.2 char* DDS_String_dup (const char * *str*)

Clone a string. Creates a new string that duplicates the value of `string`.

A string created by this method must be deleted using **DDS_String_free()** (p. 459)

Parameters:

str *<<**in**>>* *(p. 200)* The string to duplicate.

Returns:

If `string` == NULL, this method always returns NULL. Otherwise, upon success it returns a newly created string whose value is `string`; upon failure it returns NULL.

5.147.4.3 void DDS_String_free (char * *str*)

Delete a string.

Precondition:

`string` must be either NULL, or must have been created using **DDS_String_alloc()** (p. 458), **DDS_String_dup()** (p. 459)

Parameters:

str *<<**in**>>* *(p. 200)* The string to delete.

Examples:

HelloWorld.cxx.

5.147.4.4 DDS_Wchar* DDS_Wstring_alloc (DDS_UnsignedLong *length*)

Create a new empty string that can hold up to `length` wide characters.

A string created by this method must be deleted using **DDS_Wstring_free()** (p. 462)

This function will allocate enough memory to hold a string of `length` characters, **plus** one additional wide character to hold the NULL terminator.

Parameters:

length <<*in*>> *(*p. 200*)* Capacity of the string.

Returns:

A newly created non-NULL string upon success or NULL upon failure.

5.147.4.5 DDS_UnsignedLong DDS_Wstring_length (const DDS_Wchar * *str*)

Get the number of wide characters in the given string.

The result does not count the terminating zero character.

Parameters:

str <<*in*>> *(*p. 200*)* A non-NULL string.

Returns:

The number of wide characters in the string.

5.147.4.6 DDS_Wchar* DDS_Wstring_copy (DDS_Wchar * *dst*, const DDS_Wchar * *src*)

Copy the source string over the destination string reallocating the space if it's necessary.

Parameters:

dst

src

Returns:

`dst`

5.147.4.7 DDS_Wchar* DDS_Wstring_copy_and_widen (DDS_Wchar * *dst*, const char * *src*)

Copy the source string over the destination string, widening each character.

Parameters:

dst <<*in*>> *(*p. 200*)* A non-NULL string to be overwritten by `src`.

src *<<in>> (p. 200)* A non-NULL string to be copied over `dst`

Returns:

`dst`

5.147.4.8 DDS_Wchar* DDS_Wstring_dup (const DDS_Wchar * *str*)

Clone a string of wide characters. Creates a new string that duplicates the value of `string`.

A string created by this method must be deleted using **DDS_Wstring_free()** (p. 462).

Parameters:

str *<<in>> (p. 200)* The string to duplicate.

Returns:

If `string` == NULL, this method always returns NULL. Otherwise, upon success it returns a newly created string whose value is `string`; upon failure it returns NULL.

5.147.4.9 DDS_Wchar* DDS_Wstring_dup_and_widen (const char * *str*)

Clone a string of characters as a string of wide characters.

A string created by this method must be deleted using **DDS_Wstring_free()** (p. 462)

Parameters:

str *<<in>> (p. 200)* The string to duplicate.

Returns:

If `string` == NULL, this method always returns NULL. Otherwise, upon success it returns a newly created string whose value is `string`; upon failure it returns NULL.

5.147.4.10 void DDS_Wstring_free (DDS_Wchar * *str*)

Delete a string.

Precondition:

`string` must either NULL, or must have been created using **DDS_Wstring_alloc()** (p. 459), **DDS_Wstring_dup()** (p. 461), or DDS_Wstring_replace()

Parameters:

str <<***in***>> *(p. 200)* The string to delete.

Chapter 6

Class Documentation

6.1 DDS_AckResponseData_t Struct Reference

Data payload of an application-level acknowledgment.

Public Attributes

- ^ struct **DDS_OctetSeq value**
  a sequence of octets

6.1.1 Detailed Description

Data payload of an application-level acknowledgment.

6.1.2 Member Data Documentation

6.1.2.1 struct DDS_OctetSeq DDS_AckResponseData_t::value [read]

a sequence of octets

[**default**] empty (zero-length)

[**range**] Octet sequence of length [0, **DDS_DataReaderResourceLimitsQosPolicy::max_app_ack_response_length** (p. 533)],

6.2 DDS_AllocationSettings_t Struct Reference

Resource allocation settings.

Public Attributes

- **DDS_Long initial_count**

  The initial count of resources.

- **DDS_Long max_count**

  The maximum count of resources.

- **DDS_Long incremental_count**

  The incremental count of resources.

6.2.1 Detailed Description

Resource allocation settings.

QoS:

DDS_DomainParticipantResourceLimitsQosPolicy (p. 593)

6.2.2 Member Data Documentation

6.2.2.1 DDS_Long DDS_AllocationSettings_t::initial_count

The initial count of resources.

The initial resources to be allocated.

[default] It depends on the case.

[range] [0, 1 million], < max_count, (or = max_count only if increment_count == 0)

6.2.2.2 DDS_Long DDS_AllocationSettings_t::max_count

The maximum count of resources.

The maximum resources to be allocated.

[default] Depends on the case.

[**range**] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), > initial_count (or = initial_count only if increment_count == 0)

6.2.2.3 DDS_Long DDS_AllocationSettings_t::incremental_count

The incremental count of resources.

The resource to be allocated when more resources are needed.

[**default**] Depends on the case.

[**range**] -1 (Double the amount of extra memory allocated each time memory is needed) or [1,1 million] (or = 0 only if initial_count == max_count)

6.3 DDS_AsynchronousPublisherQosPolicy Struct Reference

Configures the mechanism that sends user data in an external middleware thread.

Public Attributes

- **DDS_Boolean disable_asynchronous_write**

  Disable asynchronous publishing.

- struct **DDS_ThreadSettings_t thread**

  Settings of the publishing thread.

- **DDS_Boolean disable_asynchronous_batch**

  Disable asynchronous batch flushing.

- struct **DDS_ThreadSettings_t asynchronous_batch_thread**

  Settings of the batch flushing thread.

6.3.1 Detailed Description

Configures the mechanism that sends user data in an external middleware thread.

Specifies the asynchronous publishing and asynchronous batch flushing settings of the **DDSPublisher** (p. 1346) instances.

The QoS policy specifies whether asynchronous publishing and asynchronous batch flushing are enabled for the **DDSDataWriter** (p. 1113) entities belonging to this **DDSPublisher** (p. 1346). If so, the publisher will spawn up to two threads, one for asynchronous publishing and one for asynchronous batch flushing.

See also:

DDS_BatchQosPolicy (p. 476).
DDS_PublishModeQosPolicy (p. 853).

Entity:

DDSPublisher (p. 1346)

Properties:

RxO (p. 340) = N/A

Changeable (p. 340) = **NO** (p. 340)

6.3.2 Usage

You can use this QoS policy to reduce the amount of time your application thread spends sending data.

You can also use it, along with **DDS_PublishModeQosPolicy** (p. 853) and a **DDSFlowController** (p. 1259), to send large data reliably. "Large" in this context means that the data that cannot be sent as a single packet by a network transport. For example, to send data larger than 63K reliably using UDP/IP, you must configure RTI Connext to fragment the data and send it asynchronously.

The asynchronous *publisher* thread is shared by all **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) **DDSDataWriter** (p. 1113) instances that belong to this publisher and handles their data transmission chores.

The asynchronous *batch flushing* thread is shared by all **DDSDataWriter** (p. 1113) instances with batching enabled that belong to this publisher.

This QoS policy also allows you to adjust the settings of the asynchronous publishing and the asynchronous batch flushing threads. To use different threads for two different **DDSDataWriter** (p. 1113) entities, the instances must belong to different **DDSPublisher** (p. 1346) instances.

A **DDSPublisher** (p. 1346) must have asynchronous publishing enabled for its **DDSDataWriter** (p. 1113) instances to write asynchronously.

A **DDSPublisher** (p. 1346) must have asynchronous batch flushing enabled in order to flush the batches of its **DDSDataWriter** (p. 1113) instances asynchronously. However, no asynchronous batch flushing thread will be started until the first **DDSDataWriter** (p. 1113) instance with batching enabled is created from this **DDSPublisher** (p. 1346).

6.3.3 Member Data Documentation

6.3.3.1 DDS_Boolean DDS_AsynchronousPublisherQosPolicy::disable_asynchronous_write

Disable asynchronous publishing.

If set to **DDS_BOOLEAN_TRUE** (p. 298), any **DDSDataWriter** (p. 1113) created with **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

[default] DDS_BOOLEAN_FALSE (p. 299)

6.3.3.2 struct DDS_ThreadSettings_t DDS_AsynchronousPublisherQosPolicy::thread [read]

Settings of the publishing thread.

There is only one asynchronous publishing thread per **DDSPublisher** (p. 1346).

[default] priority below normal.

The actual value depends on your architecture:

For Windows: -2

For Solaris: OS default priority

For Linux: OS default priority

For LynxOS: 13

For Integrity: 80

For VxWorks: 110

For all others: OS default priority.

[default] The actual value depends on your architecture:

For Windows: OS default stack size

For Solaris: OS default stack size

For Linux: OS default stack size

For LynxOS: 4∗16∗1024

For Integrity: 4∗20∗1024

For VxWorks: 4∗16∗1024

For all others: OS default stack size.

[default] mask = **DDS_THREAD_SETTINGS_KIND_MASK_DEFAULT** (p. 329)

6.3.3.3 DDS_Boolean DDS_AsynchronousPublisherQosPolicy::disable_asynchronous_batch

Disable asynchronous batch flushing.

If set to **DDS_BOOLEAN_TRUE** (p. 298), any **DDSDataWriter** (p. 1113) created with batching enabled will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

If **DDS_BatchQosPolicy::max_flush_delay** (p. 478) is different than **DDS_DURATION_INFINITE** (p. 305), **DDS_AsynchronousPublisherQosPolicy::disable_asynchronous_batch** (p. 469) must be set **DDS_BOOLEAN_FALSE** (p. 299).

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.3.3.4 struct DDS_ThreadSettings_t DDS_AsynchronousPublisherQosPolicy::asynchronous_batch_thread [read]

Settings of the batch flushing thread.

There is only one asynchronous batch flushing thread per **DDSPublisher** (p. 1346).

[**default**] priority below normal.

The actual value depends on your architecture:

For Windows: -2

For Solaris: OS default priority

For Linux: OS default priority

For LynxOS: 13

For Integrity: 80

For VxWorks: 110

For all others: OS default priority.

[**default**] The actual value depends on your architecture:

For Windows: OS default stack size

For Solaris: OS default stack size

For Linux: OS default stack size

For LynxOS: 4∗16∗1024

For Integrity: 4∗20∗1024

For VxWorks: 4∗16∗1024

For all others: OS default stack size.

[default] mask = **DDS_THREAD_SETTINGS_KIND_MASK_DEFAULT** (p. 329)

6.4 DDS_AvailabilityQosPolicy Struct Reference

Configures the availability of data.

Public Attributes

- **DDS_Boolean enable_required_subscriptions**

  *Enables support for required subscriptions in a **DDSDataWriter** (p. 1113).*

- struct **DDS_Duration_t max_data_availability_waiting_time**

  Defines how much time to wait before delivering a sample to the application without having received some of the previous samples.

- struct **DDS_Duration_t max_endpoint_availability_waiting_time**

  Defines how much time to wait to discover DataWriters providing samples for the same data source (virtual GUID).

- struct **DDS_EndpointGroupSeq required_matched_endpoint_groups**

  A sequence of endpoint groups.

6.4.1 Detailed Description

Configures the availability of data.

Entity:

DDSDataReader (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **YES** (p. 340) (only on a **DDSDataWriter** (p. 1113) except for the member **DDS_AvailabilityQosPolicy::enable_required_subscriptions** (p. 473))

6.4.2 Usage

This QoS policy is used in the context of two features:

- Collaborative DataWriters

- Required Subscriptions

Collaborative DataWriters

The Collaborative DataWriters feature allows having multiple DataWriters publishing samples from a common logical data source. The DataReaders will combine the samples coming from the DataWriters in order to reconstruct the correct order at the source.

This QoS policy allows you to configure the ordering and combination process in the DataReader and can be used to support two different use cases:

- **Ordered delivery of samples with RTI Persistence Service:** When a late-joining DataReader configured with **DDS_DurabilityQosPolicy** (p. 614) set to **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) or **DDS_TRANSIENT_DURABILITY_QOS** (p. 349) joins a DDS domain, it will start receiving historical samples from multiple DataWriters. For example, if the original DataWriter is still alive, the newly created DataReader will receive samples from the original DataWriter and one or more RTI Persistence Service DataWriters (PRSTDataWriters). This policy can be used to configure the sample ordering process on the DataReader.

- **Ordered delivery of samples with Group Ordered Access:** This policy can also be used to configure the sample ordering process when the Subscriber is configured with **DDS_PresentationQosPolicy** (p. 823) access_scope set to **DDS_GROUP_PRESENTATION_QOS** (p. 352). In this case, the Subscriber must deliver in order the samples published by a group of DataWriters that belong to the same Publisher and have access_scope set to **DDS_GROUP_PRESENTATION_QOS** (p. 352).

Each sample published in a DDS domain for a given logical data source is uniquely identified by a pair (virtual GUID, virtual sequence number). Samples from the same data source (same virtual GUID) can be published by different DataWriters. A DataReader will deliver a sample (VGUIDn, VSNm) to the application if one of the following conditions is satisfied:

- (VGUIDn, VSNm-1) has already been delivered to the application.

- All the known DataWriters publishing VGUIDn have announced that they do not have (VGUIDn, VSNm-1).

- None of the known DataWriters publishing GUIDn have announced potential availability of (VGUIDn, VSNm-1) and both timeouts in this QoS policy have expired.

A DataWriter announces potential availability of samples by using virtual heartbeats (HBs).

When **DDS_PresentationQosPolicy::access_scope** (p. 826) is set to **DDS_TOPIC_PRESENTATION_QOS** (p. 352) or **DDS_INSTANCE_PRESENTATION_QOS** (p. 352), the virtual HB contains information about the samples contained in the **DDSDataWriter** (p. 1113) history.

When **DDS_PresentationQosPolicy::access_scope** (p. 826) is set to **DDS_GROUP_PRESENTATION_QOS** (p. 352), the virtual HB contains information about all DataWriters in the **DDSPublisher** (p. 1346).

The frequency at which virtual HBs are sent is controlled by the protocol parameters **DDS_RtpsReliableWriterProtocol_t::virtual_heartbeat_period** (p. 894) and **DDS_RtpsReliableWriterProtocol_t::samples_per_virtual_heartbeat** (p. 895).

Required Subscriptions

In the context of Required Subscriptions, this QoS policy can be used to configure a set of Required Subscriptions on a **DDSDataWriter** (p. 1113).

Required subscriptions are preconfigured, named subscriptions that may leave and subsequently rejoin the network from time to time, at the same or different physical locations. Any time a required subscription is disconnected, any samples that would have been delivered to it are stored for delivery if and when the subscription rejoins the network.

6.4.3 Member Data Documentation

6.4.3.1 DDS_Boolean DDS_AvailabilityQosPolicy::enable_required_subscriptions

Enables support for required subscriptions in a **DDSDataWriter** (p. 1113).

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.4.3.2 struct DDS_Duration_t DDS_AvailabilityQosPolicy::max_data_availability_waiting_time [read]

Defines how much time to wait before delivering a sample to the application without having received some of the previous samples.

Collaborative DataWriters

A sample identified by (VGUIDn, VSNm) will be delivered to the application if this timeout expires for the sample and the following two conditions are satisfied:

- None of the known DataWriters publishing VGUIDn have announced po-

tential availability of (VGUIDn, VSNm-1).

- The DataWriters for all the endpoint groups specified in **required_matched_endpoint_groups** (p. 474) have been discovered or **max_endpoint_availability_waiting_time** (p. 474) has expired.

Required Subscriptions

This field is not applicable to Required Subscriptions.

[default] **DDS_DURATION_AUTO** (p. 305) (**DDS_DURATION_INFINITE** (p. 305) for **DDS_GROUP_PRESENTATION_QOS** (p. 352). Otherwise, 0 seconds)

[range] [0, **DDS_DURATION_INFINITE** (p. 305)], **DDS_DURATION_AUTO** (p. 305)

6.4.3.3 struct DDS_Duration_t DDS_AvailabilityQosPolicy::max_endpoint_availability_waiting_time [read]

Defines how much time to wait to discover DataWriters providing samples for the same data source (virtual GUID).

Collaborative DataWriters

The set of endpoint groups that are required to provide samples for a data source can be configured using **required_matched_endpoint_groups** (p. 474).

A non-consecutive sample identified by (VGUIDn, VSNm) cannot be delivered to the application unless DataWriters for all the endpoint groups in **required_matched_endpoint_groups** (p. 474) are discovered or this timeout expires.

Required Subscriptions

This field is not applicable to Required Subscriptions.

[default] **DDS_DURATION_AUTO** (p. 305) (**DDS_DURATION_INFINITE** (p. 305) for **DDS_GROUP_PRESENTATION_QOS** (p. 352). Otherwise, 0 seconds)

[range] [0, **DDS_DURATION_INFINITE** (p. 305)], **DDS_DURATION_AUTO** (p. 305)

6.4.3.4 struct DDS_EndpointGroupSeq DDS_AvailabilityQosPolicy::required_matched_endpoint_groups [read]

A sequence of endpoint groups.

Collaborative DataWriters

In the context of Collaborative DataWriters, it specifies the set of endpoint groups that are expected to provide samples for the same data source.

The quorum count in a group represents the number of DataWriters that must be discovered for that group before the DataReader is allowed to provide non consecutive samples to the application.

A DataWriter becomes a member of an endpoint group by configuring the role_name in **DDS_DataWriterQos::publication_name** (p. 559).

Required Subscriptions

In the context of Required Subscriptions, it specifies the set of Required Subscriptions on a **DDSDataWriter** (p. 1113).

Each Required Subscription is specified by a name and a quorum count.

The quorum count represents the number of DataReaders that have to acknowledge the sample before it can be considered fully acknowledged for that Required Subscription.

A DataReader is associated with a Required Subscription by configuring the role_name in **DDS_DataReaderQos::subscription_name** (p. 520).

[**default**] Empty sequence

6.5 DDS_BatchQosPolicy Struct Reference

Used to configure batching of multiple samples into a single network packet in order to increase throughput for small samples.

Public Attributes

- **DDS_Boolean enable**

  Specifies whether or not batching is enabled.

- **DDS_Long max_data_bytes**

  The maximum cumulative length of all serialized samples in a batch.

- **DDS_Long max_samples**

  The maximum number of samples in a batch.

- struct **DDS_Duration_t max_flush_delay**

  The maximum flush delay.

- struct **DDS_Duration_t source_timestamp_resolution**

  Batch source timestamp resolution.

- **DDS_Boolean thread_safe_write**

  Determines whether or not the write operation is thread safe.

6.5.1 Detailed Description

Used to configure batching of multiple samples into a single network packet in order to increase throughput for small samples.

This QoS policy configures the ability of the middleware to collect multiple user data samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput.

This QoS policy can be used to dramatically increase effective throughput for small data samples. Usually, throughput for small samples (size $<$ 2048 bytes) is limited by CPU capacity and not by network bandwidth. Batching many smaller samples to be sent in a single large packet will increase network utilization, and thus throughput, in terms of samples per second.

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.5.2 Member Data Documentation

6.5.2.1 DDS_Boolean DDS_BatchQosPolicy::enable

Specifies whether or not batching is enabled.

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.5.2.2 DDS_Long DDS_BatchQosPolicy::max_data_bytes

The maximum cumulative length of all serialized samples in a batch.

A batch is flushed automatically when this maximum is reached.

max_data_bytes does not include the meta data associated with the batch samples. Each sample has at least 8 bytes of meta data containing information such as the timestamp and sequence number. The meta data can be as large as 52 bytes for keyed topics and 20 bytes for unkeyed topics.

Note: Batches must contain whole samples. If a new batch is started and its initial sample causes the serialized size to exceed max_data_bytes, RTI Connext will send the sample in a single batch.

[**default**] 1024

[**range**] [1,**DDS_LENGTH_UNLIMITED** (p. 371)]

6.5.3 Consistency

The setting of **DDS_BatchQosPolicy::max_data_bytes** (p. 477) must be consistent with **DDS_BatchQosPolicy::max_samples** (p. 477). For these two values to be consistent, they cannot be both **DDS_LENGTH_UNLIMITED** (p. 371).

6.5.3.1 DDS_Long DDS_BatchQosPolicy::max_samples

The maximum number of samples in a batch.

A batch is flushed automatically when this maximum is reached.

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

[**range**] [1,**DDS_LENGTH_UNLIMITED** (p. 371)]

6.5.4 Consistency

The setting of **DDS_BatchQosPolicy::max_samples** (p. 477) must be consistent with **DDS_BatchQosPolicy::max_data_bytes** (p. 477). For these two values to be consistent, they cannot be both **DDS_LENGTH_UNLIMITED** (p. 371).

6.5.4.1 struct DDS_Duration_t DDS_BatchQosPolicy::max_flush_delay [read]

The maximum flush delay.

A batch is flushed automatically after the delay specified by this parameter.

The delay is measured from the time the first sample in the batch is written by the application.

[default] **DDS_DURATION_INFINITE** (p. 305)

[range] [0,**DDS_DURATION_INFINITE** (p. 305)]

6.5.5 Consistency

The setting of **DDS_BatchQosPolicy::max_flush_delay** (p. 478) must be consistent with **DDS_AsynchronousPublisherQosPolicy::disable_asynchronous_batch** (p. 469) and **DDS_BatchQosPolicy::thread_safe_write** (p. 479). If the delay is different than **DDS_DURATION_INFINITE** (p. 305), **DDS_AsynchronousPublisherQosPolicy::disable_asynchronous_batch** (p. 469) must be set to **DDS_BOOLEAN_FALSE** (p. 299) and **DDS_BatchQosPolicy::thread_safe_write** (p. 479) must be set to **DDS_BOOLEAN_TRUE** (p. 298).

6.5.5.1 struct DDS_Duration_t DDS_BatchQosPolicy::source_timestamp_resolution [read]

Batch source timestamp resolution.

The value of this field determines how the source timestamp is associated with the samples in a batch.

A sample written with timestamp 't' inherits the source timestamp 't2' associated with the previous sample unless ('t' - 't2') > source_timestamp_resolution.

If source_timestamp_resolution is set to **DDS_DURATION_INFINITE** (p. 305), every sample in the batch will share the source timestamp associated with the first sample.

If source_timestamp_resolution is set to zero, every sample in the batch will contain its own source timestamp corresponding to the moment when the sample was written.

The performance of the batching process is better when source_timestamp_resolution is set to **DDS_DURATION_INFINITE** (p. 305).

[**default**] **DDS_DURATION_INFINITE** (p. 305)

[**range**] [0,**DDS_DURATION_INFINITE** (p. 305)]

6.5.6 Consistency

The setting of **DDS_BatchQosPolicy::source_timestamp_resolution** (p. 478) must be consistent with **DDS_BatchQosPolicy::thread_safe_write** (p. 479). If **DDS_BatchQosPolicy::thread_safe_write** (p. 479) is set to **DDS_BOOLEAN_FALSE** (p. 299), **DDS_BatchQosPolicy::source_timestamp_resolution** (p. 478) must be set to **DDS_DURATION_INFINITE** (p. 305).

6.5.6.1 DDS_Boolean DDS_BatchQosPolicy::thread_safe_write

Determines whether or not the write operation is thread safe.

If this parameter is set to **DDS_BOOLEAN_TRUE** (p. 298), multiple threads can call write on the **DDSDataWriter** (p. 1113) concurrently.

[**default**] **DDS_BOOLEAN_TRUE** (p. 298)

6.5.7 Consistency

The setting of **DDS_BatchQosPolicy::thread_safe_write** (p. 479) must be consistent with **DDS_BatchQosPolicy::source_timestamp_resolution** (p. 478). If **DDS_BatchQosPolicy::thread_safe_write** (p. 479) is set to **DDS_BOOLEAN_FALSE** (p. 299), **DDS_BatchQosPolicy::source_timestamp_resolution** (p. 478) must be set to **DDS_DURATION_INFINITE** (p. 305).

6.6 DDS_BooleanSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_Boolean** (p. 301) >.

6.6.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_Boolean** (p. 301) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Boolean (p. 301)
FooSeq (p. 1494)

6.7 DDS_BuiltinTopicKey_t Struct Reference

The key type of the built-in topic types.

Public Attributes

- DDS_BUILTIN_TOPIC_KEY_TYPE_NATIVE **value** [DDS_BUILTIN_TOPIC_KEY_TYPE_NATIVE_LENGTH]

  *An array of four integers that uniquely represents a remote **DDSEntity** (p. 1253).*

6.7.1 Detailed Description

The key type of the built-in topic types.

Each remote **DDSEntity** (p. 1253) to be discovered is can be uniquely identified by this key. This is the key of all the built-in topic data types.

See also:

DDS_ParticipantBuiltinTopicData (p. 816)
DDS_TopicBuiltinTopicData (p. 958)
DDS_PublicationBuiltinTopicData (p. 839)
DDS_SubscriptionBuiltinTopicData (p. 936)

6.7.2 Member Data Documentation

6.7.2.1 DDS_BUILTIN_TOPIC_KEY_TYPE_NATIVE DDS_BuiltinTopicKey_t::value[DDS_BUILTIN_TOPIC_KEY_TYPE_NATIVE_LENGTH]

An array of four integers that uniquely represents a remote **DDSEntity** (p. 1253).

6.8 DDS_BuiltinTopicReaderResourceLimits_t Struct Reference

Built-in topic reader's resource limits.

Public Attributes

- **DDS_Long initial_samples**

  Initial number of samples.

- **DDS_Long max_samples**

  Maximum number of samples.

- **DDS_Long initial_infos**

  Initial number of sample infos.

- **DDS_Long max_infos**

  Maximum number of sample infos.

- **DDS_Long initial_outstanding_reads**

  The initial number of outstanding reads that have not call finish yet on the same built-in topic ***DDSDataReader*** *(*p. 1087*).*

- **DDS_Long max_outstanding_reads**

  The maximum number of outstanding reads that have not called finish yet on the same built-in topic ***DDSDataReader*** *(*p. 1087*).*

- **DDS_Long max_samples_per_read**

  Maximum number of samples that can be read/taken on a same built-in topic ***DDSDataReader*** *(*p. 1087*).*

- **DDS_Boolean disable_fragmentation_support**
- **DDS_Long max_fragmented_samples**
- **DDS_Long initial_fragmented_samples**
- **DDS_Long max_fragmented_samples_per_remote_writer**
- **DDS_Long max_fragments_per_sample**
- **DDS_Boolean dynamically_allocate_fragmented_samples**

6.8.1 Detailed Description

Built-in topic reader's resource limits.

Defines the resources that can be used for a built-in-topic data reader.

A built-in topic data reader subscribes reliably to built-in topics containing declarations of new entities or updates to existing entities in the domain. Keys are used to differentiate among entities of the same type. RTI Connext assigns a unique key to each entity in a domain.

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

QoS:

DDS_DiscoveryConfigQosPolicy (p. 573)

6.8.2 Member Data Documentation

6.8.2.1 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::initial_samples

Initial number of samples.

This should be a value between 1 and initial number of instance of the built-in-topic reader, depending on how many instances are sending data concurrently.

[**default**] 64

[**range**] [1, 1 million], <= max_samples

6.8.2.2 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_samples

Maximum number of samples.

This should be a value between 1 and max number of instance of the built-in-topic reader, depending on how many instances are sending data concurrently. Also, it should not be less than initial_samples.

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

[**range**] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_samples

6.8.2.3 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::initial_infos

Initial number of sample infos.

The initial number of info units that a built-in topic **DDSDataReader** (p. 1087) can have. Info units are used to store **DDS_SampleInfo** (p. 912).

[default] 64

[range] [1, 1 million] <= max_infos

6.8.2.4 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_infos

Maximum number of sample infos.

The maximum number of info units that a built-in topic **DDSDataReader** (p. 1087) can use to store **DDS_SampleInfo** (p. 912).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_infos

6.8.2.5 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::initial_outstanding_reads

The initial number of outstanding reads that have not call finish yet on the same built-in topic **DDSDataReader** (p. 1087).

[default] 2

[range] [1, 65536] or **DDS_LENGTH_UNLIMITED** (p. 371) <= max_outstanding_reads

6.8.2.6 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_outstanding_reads

The maximum number of outstanding reads that have not called finish yet on the same built-in topic **DDSDataReader** (p. 1087).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 65536] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_outstanding_reads

6.8.2.7 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_samples_per_read

Maximum number of samples that can be read/taken on a same built-in topic **DDSDataReader** (p. 1087).

[**default**] 1024

[**range**] [1, 65536]

6.8.2.8 DDS_Boolean DDS_BuiltinTopicReaderResourceLimits_t::disable_fragmentation_support

6.8.2.9 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_fragmented_samples

6.8.2.10 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::initial_fragmented_samples

6.8.2.11 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_fragmented_samples_per_remote_writer

6.8.2.12 DDS_Long DDS_BuiltinTopicReaderResourceLimits_t::max_fragments_per_sample

6.8.2.13 DDS_Boolean DDS_BuiltinTopicReaderResourceLimits_t::dynamically_allocate_fragmented_samples

6.9 DDS_ChannelSettings_t Struct Reference

Type used to configure the properties of a channel.

Public Attributes

- struct **DDS_TransportMulticastSettingsSeq multicast_settings**

  *A sequence of **DDS_TransportMulticastSettings_t** (p. 980) used to configure the multicast addresses associated with a channel.*

- char * **filter_expression**

  A logical expression used to determine the data that will be published in the channel.

- **DDS_Long priority**

  Publication priority.

6.9.1 Detailed Description

Type used to configure the properties of a channel.

QoS:

DDS_MultiChannelQosPolicy (p. 796)

6.9.2 Member Data Documentation

6.9.2.1 struct DDS_TransportMulticastSettingsSeq DDS_ChannelSettings_t::multicast_settings [read]

A sequence of **DDS_TransportMulticastSettings_t** (p. 980) used to configure the multicast addresses associated with a channel.

The sequence cannot be empty.

The maximum number of multicast locators in a channel is limited to four (A locator is defined by a transport alias, a multicast address and a port)

[default] Empty sequence (invalid value)

6.9.2.2 char* DDS_ChannelSettings_t::filter_expression

A logical expression used to determine the data that will be published in the channel.

If the expression evaluates to TRUE, a sample will be published on the channel.

An empty string always evaluates the expression to TRUE.

A NULL value is not allowed.

The syntax of the expression will depend on the value of **DDS_MultiChannelQosPolicy::filter_name** (p. 798)

Important: This value must be an allocated string with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459). It should not be assigned to a string constant.

The filter expression length (including NULL-terminated character) cannot be greater than **DDS_DomainParticipantResourceLimitsQosPolicy::channel_filter_expression_max_length** (p. 610).

See also:

Queries and Filters Syntax (p. 208)

[**default**] NULL (invalid value)

6.9.2.3 DDS_Long DDS_ChannelSettings_t::priority

Publication priority.

A positive integer value designating the relative priority of the channel, used to determine the transmission order of pending writes.

Use of publication priorities requires the asynchronous publisher (**DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422)) with **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) set to **DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90).

Larger numbers have higher priority.

If the publication priority of the channel is any value other than **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), then the channel's priority will take precedence over the data writer's priority.

If the publication priority of the channel is set to **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), then the channel's priority will be set to the value of the data writer's priority.

If the publicaton priority of both the data writer and the channel are **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), the channel will be assigned the lowest priority value.

If the publication priority of the channel is **DDS_PUBLICATION_PRIORITY_AUTOMATIC** (p. 421), then the channel will be assigned the

priority of the largest publication priority of all samples in the channel. The publication priority of each sample can be set in the **DDS_WriteParams_t** (p. 1067) of the **FooDataWriter::write_w_params** (p. 1487) function.

[default] DDS_PUBLICATION_PRIORITY_UNDEFINED (p. 421)

6.10 DDS_ChannelSettingsSeq Struct Reference

Declares IDL `sequence<` **DDS_ChannelSettings_t** (p. 486) `>`.

6.10.1 Detailed Description

Declares IDL `sequence<` **DDS_ChannelSettings_t** (p. 486) `>`.

A sequence of **DDS_ChannelSettings_t** (p. 486) used to configure the channels' properties. If the length of the sequence is zero, the **DDS_MultiChannelQosPolicy** (p. 796) has no effect.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_ChannelSettings_t (p. 486)

6.11 DDS_CharSeq Struct Reference

Instantiates FooSeq (p. 1494) < **DDS_Char** (p. 299) >.

6.11.1 Detailed Description

Instantiates FooSeq (p. 1494) < **DDS_Char** (p. 299) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Char (p. 299)
FooSeq (p. 1494)

6.12 DDS_ContentFilterProperty_t Struct Reference

<<*eXtension*>> *(p. 199)* Type used to provide all the required information to enable content filtering.

Public Attributes

- char * **content_filter_topic_name**

  Name of the Content-filtered Topic associated with the Reader.

- char * **related_topic_name**

  Name of the Topic related to the Content-filtered Topic.

- char * **filter_class_name**

  Identifies the filter class this filter belongs to. RTPS can support multiple filter classes (SQL, regular expressions, custom filters, etc).

- char * **filter_expression**

  The actual filter expression. Must be a valid expression for the filter class specified using filterClassName.

- struct **DDS_StringSeq expression_parameters**

  Defines the value for each parameter in the filter expression.

6.12.1 Detailed Description

<<*eXtension*>> *(p. 199)* Type used to provide all the required information to enable content filtering.

6.12.2 Member Data Documentation

6.12.2.1 char* DDS_ContentFilterProperty_t::content_filter_topic_name

Name of the Content-filtered Topic associated with the Reader.

6.12.2.2 char* DDS_ContentFilterProperty_t::related_topic_name

Name of the Topic related to the Content-filtered Topic.

6.12.2.3 char* DDS_ContentFilterProperty_t::filter_class_name

Identifies the filter class this filter belongs to. RTPS can support multiple filter classes (SQL, regular expressions, custom filters, etc).

6.12.2.4 char* DDS_ContentFilterProperty_t::filter_expression

The actual filter expression. Must be a valid expression for the filter class specified using filterClassName.

6.12.2.5 struct DDS_StringSeq DDS_ContentFilterProperty_t::expression_parameters [read]

Defines the value for each parameter in the filter expression.

6.13 DDS_Cookie_t Struct Reference

<<*eXtension*>> *(*p. 199*)* Sequence of bytes identifying a written data sample, used when writing with parameters.

Public Attributes

- struct **DDS_OctetSeq value**

  a sequence of octets

6.13.1 Detailed Description

<<*eXtension*>> *(*p. 199*)* Sequence of bytes identifying a written data sample, used when writing with parameters.

6.13.2 Member Data Documentation

6.13.2.1 struct DDS_OctetSeq DDS_Cookie_t::value [read]

a sequence of octets

[**default**] Empty (zero-sized)

[**range**] Octet sequence of length [0,DDS_DataWriterResourceLimitsQosPolicy::cookie_max_length]

6.14 DDS_CookieSeq Struct Reference

6.14.1 Detailed Description

6.15 DDS_DatabaseQosPolicy Struct Reference

Various threads and resource limits settings used by RTI Connext to control its internal database.

Public Attributes

- struct **DDS_ThreadSettings_t thread**

  Database thread settings.

- struct **DDS_Duration_t shutdown_timeout**

  The maximum wait time during a shutdown.

- struct **DDS_Duration_t cleanup_period**

  The database thread will wake up at this rate to clean up the database.

- struct **DDS_Duration_t shutdown_cleanup_period**

  The clean-up period used during database shut-down.

- **DDS_Long initial_records**

  The initial number of total records.

- **DDS_Long max_skiplist_level**

  The maximum level of the skiplist.

- **DDS_Long max_weak_references**

  The maximum number of weak references.

- **DDS_Long initial_weak_references**

  The initial number of weak references.

6.15.1 Detailed Description

Various threads and resource limits settings used by RTI Connext to control its internal database.

RTI uses an internal in-memory "database" to store information about entities created locally as well as remote entities found during the discovery process. This database uses a background thread to garbage-collect records related to deleted entities. When the **DDSDomainParticipant** (p. 1139) that maintains this database is deleted, it shuts down this thread.

The Database QoS policy is used to configure how RTI Connext manages its database, including how often it cleans up, the priority of the database thread, and limits on resources that may be allocated by the database.

You may be interested in modifying the **DDS_DatabaseQosPolicy::shutdown_timeout** (p. 497) and **DDS_DatabaseQosPolicy::shutdown_cleanup_period** (p. 497) parameters to decrease the time it takes to delete a **DDSDomainParticipant** (p. 1139) when your application is shutting down.

The **DDS_DomainParticipantResourceLimitsQosPolicy** (p. 593) controls the memory allocation for elements stored in the database.

This QoS policy is an extension to the DDS standard.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) **NO** (p. 340)

6.15.2 Member Data Documentation

6.15.2.1 struct DDS_ThreadSettings_t DDS_DatabaseQosPolicy::thread [read]

Database thread settings.

There is only one database thread: the clean-up thread.

[default] priority low.

The actual value depends on your architecture:

For Windows: -3

For Solaris: OS default priority

For Linux: OS default priority

For LynxOS: 10

For INTEGRITY: 60

For VxWorks: 120

For all others: OS default priority.

[default] The actual value depends on your architecture:

For Windows: OS default stack size

For Solaris: OS default stack size

For Linux: OS default stack size

For LynxOS: 16*1024

For INTEGRITY: 20*1024

For VxWorks: 16*1024

For all others: OS default stack size.

[**default**] mask **DDS_THREAD_SETTINGS_STDIO** (p. 329)

6.15.2.2 struct DDS_Duration_t DDS_DatabaseQosPolicy::shutdown_timeout [read]

The maximum wait time during a shutdown.

The domain participant will exit after the timeout, even if the database has not been fully cleaned up.

[**default**] 15 seconds

[**range**] [0,**DDS_DURATION_INFINITE** (p. 305)]

6.15.2.3 struct DDS_Duration_t DDS_DatabaseQosPolicy::cleanup_period [read]

The database thread will wake up at this rate to clean up the database.

[**default**] 61 seconds

[**range**] [0,1 year]

6.15.2.4 struct DDS_Duration_t DDS_DatabaseQosPolicy::shutdown_cleanup_period [read]

The clean-up period used during database shut-down.

[**default**] 1 second

[**range**] [0,1 year]

6.15.2.5 DDS_Long DDS_DatabaseQosPolicy::initial_records

The initial number of total records.

[default] 1024

[range] [1,10 million]

6.15.2.6 DDS_Long DDS_DatabaseQosPolicy::max_skiplist_level

The maximum level of the skiplist.

The skiplist is used to keep records in the database. Usually, the search time is log2(N), where N is the total number of records in one skiplist. However, once N exceeds 2^n, where n is the maximum skiplist level, the search time will become more and more linear. Therefore, the maximum level should be set such that 2^n is larger than the maximum(N among all skiplists). Usually, the maximum N is the maximum number of remote and local writers or readers.

[default] 14

[range] [1,31]

6.15.2.7 DDS_Long DDS_DatabaseQosPolicy::max_weak_references

The maximum number of weak references.

A weak reference is an internal data structure that refers to a record within RTI Connext' internal database. This field configures the maximum number of such references that RTI Connext may create.

The actual number of weak references is permitted to grow from an initial value (indicated by **DDS_DatabaseQosPolicy::initial_weak_references** (p. 499)) to this maximum. To prevent RTI Connext from allocating any weak references after the system has reached a steady state, set the initial and maximum values equal to one another. To indicate that the number of weak references should continue to grow as needed indefinitely, set this field to **DDS_LENGTH_UNLIMITED** (p. 371). Be aware that although a single weak reference occupies very little memory, allocating a very large number of them can have a significant impact on your overall memory usage.

Tuning this value precisely is difficult without intimate knowledge of the structure of RTI Connext' database; doing so is an advanced feature not required by most applications. The default value has been chosen to be sufficient for reasonably large systems. If you believe you may need to modify this value, please consult with RTI support personnel for assistance.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 100 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_weak_references

See also:

DDS_DatabaseQosPolicy::initial_weak_references (p. 499)

6.15.2.8 DDS_Long DDS_DatabaseQosPolicy::initial_weak_references

The initial number of weak references.

See **DDS_DatabaseQosPolicy::max_weak_references** (p. 498) for more information about what a weak reference is.

If the QoS set contains an initial_weak_references value that is too small to ever grow to **DDS_DatabaseQosPolicy::max_weak_references** (p. 498) using RTI Connext' internal algorithm, this value will be adjusted upwards as necessary. Subsequent accesses of this value will reveal the actual initial value used.

Changing the value of this field is an advanced feature; it is recommended that you consult with RTI support personnel before doing so.

[**default**] 2049, which is the minimum initial value imposed by REDA when the maximum is unlimited. If a lower value is specified, it will simply be increased to 2049 automatically.

[**range**] [1, 100 million], <= max_weak_references

See also:

DDS_DatabaseQosPolicy::max_weak_references (p. 498)

6.16 DDS_DataReaderCacheStatus Struct Reference

<<*eXtension*>> *(*p. *199)* The status of the reader's cache.

Public Attributes

- **DDS_LongLong sample_count_peak**

  The highest number of samples in the reader's queue over the lifetime of the reader.

- **DDS_LongLong sample_count**

  The number of samples in the reader's queue.

6.16.1 Detailed Description

<<*eXtension*>> *(*p. *199)* The status of the reader's cache.

Entity:

DDSDataReader (p. 1087)

6.16.2 Member Data Documentation

6.16.2.1 DDS_LongLong DDS_DataReaderCacheStatus::sample_count_peak

The highest number of samples in the reader's queue over the lifetime of the reader.

6.16.2.2 DDS_LongLong DDS_DataReaderCacheStatus::sample_count

The number of samples in the reader's queue.

includes samples that may not yet be available to be read or taken by the user, due to samples being received out of order or **PRESENTATION** (p. 351)

6.17 DDS_DataReaderProtocolQosPolicy Struct Reference

Along with **DDS_WireProtocolQosPolicy** (p. 1059) and **DDS_DataWriterProtocolQosPolicy** (p. 535), this QoS policy configures the DDS on-the-network protocol (RTPS).

Public Attributes

- struct **DDS_GUID_t virtual_guid**

  The virtual GUID (Global Unique Identifier).

- **DDS_UnsignedLong rtps_object_id**

  The RTPS Object ID.

- **DDS_Boolean expects_inline_qos**

  Specifies whether this DataReader expects inline QoS with every sample.

- **DDS_Boolean disable_positive_acks**

  Whether the reader sends positive acknowledgements to writers.

- **DDS_Boolean propagate_dispose_of_unregistered_instances**

  *Indicates whether or not an instance can move to the **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) state without being in the **DDS_ALIVE_INSTANCE_STATE** (p. 117) state.*

- struct **DDS_RtpsReliableReaderProtocol_t rtps_reliable_reader**

  The reliable protocol defined in RTPS.

6.17.1 Detailed Description

Along with **DDS_WireProtocolQosPolicy** (p. 1059) and **DDS_DataWriterProtocolQosPolicy** (p. 535), this QoS policy configures the DDS on-the-network protocol (RTPS).

DDS has a standard protocol for packet (user and meta data) exchange between applications using DDS for communications. This QoS policy and **DDS_DataReaderProtocolQosPolicy** (p. 501) give you control over configurable portions of the protocol, including the configuration of the reliable data delivery mechanism of the protocol on a per DataWriter or DataReader basis.

These configuration parameters control timing, timeouts, and give you the ability to tradeoff between speed of data loss detection and repair versus network and CPU bandwidth used to maintain reliability.

It is important to tune the reliability protocol (on a per **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) basis) to meet the requirements of the end-user application so that data can be sent between DataWriters and DataReaders in an efficient and optimal manner in the presence of data loss.

You can also use this QoS policy to control how RTI Connext responds to "slow" reliable DataReaders or ones that disconnect or are otherwise lost. See **DDS_ReliabilityQosPolicy** (p. 865) for more information on the per-DataReader/DataWriter reliability configuration. **DDS_HistoryQosPolicy** (p. 758) and **DDS_ResourceLimitsQosPolicy** (p. 879) also play an important role in the DDS reliable protocol.

This QoS policy is an extension to the DDS standard.

Entity:

DDSDataReader (p. 1087)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.17.2 Member Data Documentation

6.17.2.1 struct DDS_GUID_t DDS_DataReaderProtocolQosPolicy::virtual_guid [read]

The virtual GUID (Global Unique Identifier).

The virtual GUID is used to uniquely identify different incarnations of the same **DDSDataReader** (p. 1087).

The association between a **DDSDataReader** (p. 1087) and its persisted state is done using the virtual GUID.

[default] DDS_GUID_AUTO (p. 309)

6.17.2.2 DDS_UnsignedLong DDS_DataReaderProtocolQosPolicy::rtps_object_id

The RTPS Object ID.

This value is used to determine the RTPS object ID of a data reader according to the DDS-RTPS Interoperability Wire Protocol.

Only the last 3 bytes are used; the most significant byte is ignored.

If the default value is specified, RTI Connext will automatically assign the object ID based on a counter value (per participant) starting at 0x00800000. That value is incremented for each new data reader.

A rtps_object_id value in the interval [0x00800000,0x00ffffff] may collide with the automatic values assigned by RTI Connext. In those cases, the recomendation is not to use automatic object ID assigment.

[default] DDS_RTPS_AUTO_ID (p. **??**)

[range] [0,0x00ffffff]

6.17.2.3 DDS_Boolean DDS_DataReaderProtocolQosPolicy::expects_inline_qos

Specifies whether this DataReader expects inline QoS with every sample.

In RTI Connext, a **DDSDataReader** (p. 1087) nominally relies on Discovery to propagate QoS on a matched **DDSDataWriter** (p. 1113). Alternatively, a **DDSDataReader** (p. 1087) may get information on a matched **DDSDataWriter** (p. 1113) through QoS sent inline with a sample.

Asserting **DDS_DataReaderProtocolQosPolicy::expects_inline_qos** (p. 503) indicates to a matching **DDSDataWriter** (p. 1113) that this **DDSDataReader** (p. 1087) expects to receive inline QoS with every sample. The complete set of inline QoS that a **DDSDataWriter** (p. 1113) may send inline is specified by the Real-Time Publish-Subscribe (RTPS) Wire Interoperability Protocol.

Because RTI Connext **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) cache Discovery information, inline QoS are largely redundant and thus unecessary. Only for other stateless implementations whose **DDSDataReader** (p. 1087) does not cache Discovery information is inline QoS necessary.

Also note that inline QoS are additional wire-payload that consume additional bandwidth and serialization and deserialization time.

[default] DDS_BOOLEAN_FALSE (p. 299)

6.17.2.4 DDS_Boolean DDS_DataReaderProtocolQosPolicy::disable_positive_acks

Whether the reader sends positive acknowledgements to writers.

If set to **DDS_BOOLEAN_TRUE** (p. 298), the reader does not send positive

acknowledgments (ACKs) in response to Heartbeat messages. The reader will send negative acknowledgements (NACKs) when a Heartbeat advertises samples that it has not received.

Otherwise, if set to **DDS_BOOLEAN_FALSE** (p. 299) (the default), the reader will send ACKs to writers that expect ACKs (**DDS_DataWriterProtocolQosPolicy::disable_positive_acks** (p. 537) = **DDS_BOOLEAN_FALSE** (p. 299)) and it will not send ACKs to writers that disable ACKs (**DDS_DataWriterProtocolQosPolicy::disable_positive_acks** (p. 537) = **DDS_BOOLEAN_TRUE** (p. 298))

[default] DDS_BOOLEAN_FALSE (p. 299)

6.17.2.5 DDS_Boolean DDS_DataReaderProtocolQosPolicy::propagate_dispose_of_unregistered_instances

Indicates whether or not an instance can move to the **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) state without being in the **DDS_ALIVE_INSTANCE_STATE** (p. 117) state.

This field only applies to keyed readers.

When the field is set to **DDS_BOOLEAN_TRUE** (p. 298), the DataReader will receive dispose notifications even if the instance is not alive.

To guarantee the key availability through the usage of the API **FooDataReader::get_key_value** (p. 1472), this option should be used in combination **DDS_DataWriterProtocolQosPolicy::serialize_key_with_dispose** (p. 538) on the DataWriter that should be set to **DDS_BOOLEAN_TRUE** (p. 298).

[default] DDS_BOOLEAN_FALSE (p. 299)

6.17.2.6 struct DDS_RtpsReliableReaderProtocol_t DDS_DataReaderProtocolQosPolicy::rtps_reliable_reader [read]

The reliable protocol defined in RTPS.

[default] min_heartbeat_response_delay 0 seconds; max_heartbeat_response_delay 0.5 seconds

6.18 DDS_DataReaderProtocolStatus Struct Reference

<<*eXtension*>> *(*p. *199)* The status of a reader's internal protocol related metrics, like the number of samples received, filtered, rejected; and status of wire protocol traffic.

Public Attributes

- **DDS_LongLong received_sample_count**

  The number of user samples from a remote DataWriter received for the first time by a local DataReader.

- **DDS_LongLong received_sample_count_change**

  The incremental change in the number of user samples from a remote DataWriter received for the first time by a local DataReader since the last time the status was read.

- **DDS_LongLong received_sample_bytes**

  The number of bytes of user samples from a remote DataWriter received for the first time by a local DataReader.

- **DDS_LongLong received_sample_bytes_change**

  The incremental change in the number of bytes of user samples from a remote DataWriter received for the first time by a local DataReader since the last time the status was read.

- **DDS_LongLong duplicate_sample_count**

  The number of samples from a remote DataWriter received, not for the first time, by a local DataReader.

- **DDS_LongLong duplicate_sample_count_change**

  The incremental change in the number of samples from a remote DataWriter received, not for the first time, by a local DataReader since the last time the status was read.

- **DDS_LongLong duplicate_sample_bytes**

  The number of bytes of samples from a remote DataWriter received, not for the first time, by a local DataReader.

- **DDS_LongLong duplicate_sample_bytes_change**

  The incremental change in the number of bytes of samples from a remote DataWriter received, not for the first time, by a local DataReader since the last time the status was read.

^ **DDS_LongLong filtered_sample_count**

The number of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter.

^ **DDS_LongLong filtered_sample_count_change**

The incremental change in the number of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter since the last time the status was read.

^ **DDS_LongLong filtered_sample_bytes**

The number of bytes of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter.

^ **DDS_LongLong filtered_sample_bytes_change**

The incremental change in the number of bytes of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter since the last time the status was read.

^ **DDS_LongLong received_heartbeat_count**

The number of Heartbeats from a remote DataWriter received by a local DataReader.

^ **DDS_LongLong received_heartbeat_count_change**

The incremental change in the number of Heartbeats from a remote DataWriter received by a local DataReader since the last time the status was read.

^ **DDS_LongLong received_heartbeat_bytes**

The number of bytes of Heartbeats from a remote DataWriter received by a local DataReader.

^ **DDS_LongLong received_heartbeat_bytes_change**

The incremental change in the number of bytes of Heartbeats from a remote DataWriter received by a local DataReader since the last time the status was read.

^ **DDS_LongLong sent_ack_count**

The number of ACKs sent from a local DataReader to a matching remote DataWriter.

^ **DDS_LongLong sent_ack_count_change**

The incremental change in the number of ACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

- **DDS_LongLong sent_ack_bytes**

  The number of bytes of ACKs sent from a local DataReader to a matching remote DataWriter.

- **DDS_LongLong sent_ack_bytes_change**

  The incremental change in the number of bytes of ACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

- **DDS_LongLong sent_nack_count**

  The number of NACKs sent from a local DataReader to a matching remote DataWriter.

- **DDS_LongLong sent_nack_count_change**

  The incremental change in the number of NACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

- **DDS_LongLong sent_nack_bytes**

  The number of bytes of NACKs sent from a local DataReader to a matching remote DataWriter.

- **DDS_LongLong sent_nack_bytes_change**

  The incremental change in the number of bytes of NACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

- **DDS_LongLong received_gap_count**

  The number of GAPs received from remote DataWriter to this DataReader.

- **DDS_LongLong received_gap_count_change**

  The incremental change in the number of GAPs received from remote DataWriter to this DataReader since the last time the status was read.

- **DDS_LongLong received_gap_bytes**

  The number of bytes of GAPs received from remote DataWriter to this DataReader.

- **DDS_LongLong received_gap_bytes_change**

  The incremental change in the number of bytes of GAPs received from remote DataWriter to this DataReader since the last time the status was read.

- **DDS_LongLong rejected_sample_count**

  The number of times a sample is rejected due to exceptions in the receive path.

- **DDS_LongLong rejected_sample_count_change**

  The incremental change in the number of times a sample is rejected due to exceptions in the receive path since the last time the status was read.

- struct **DDS_SequenceNumber_t first_available_sample_sequence_number**

  Sequence number of the first available sample in a matched Datawriters reliability queue.

- struct **DDS_SequenceNumber_t last_available_sample_sequence_number**

  Sequence number of the last available sample in a matched Datawriter's reliability queue.

- struct **DDS_SequenceNumber_t last_committed_sample_sequence_number**

  Sequence number of the newest sample received from the matched DataWriter committed to the DataReader's queue.

- **DDS_Long uncommitted_sample_count**

  Number of received samples that are not yet available to be read or taken, due to being received out of order.

6.18.1 Detailed Description

<<***eXtension***>> *(p. 199)* The status of a reader's internal protocol related metrics, like the number of samples received, filtered, rejected; and status of wire protocol traffic.

Entity:

DDSDataReader (p. 1087)

6.18.2 Member Data Documentation

6.18.2.1 DDS_LongLong DDS_DataReaderProtocolStatus::received_sample_count

The number of user samples from a remote DataWriter received for the first time by a local DataReader.

6.18.2.2 DDS_LongLong DDS_-DataReaderProtocolStatus::received_sample_count_change

The incremental change in the number of user samples from a remote DataWriter received for the first time by a local DataReader since the last time the status was read.

6.18.2.3 DDS_LongLong DDS_-DataReaderProtocolStatus::received_sample_bytes

The number of bytes of user samples from a remote DataWriter received for the first time by a local DataReader.

6.18.2.4 DDS_LongLong DDS_-DataReaderProtocolStatus::received_sample_bytes_change

The incremental change in the number of bytes of user samples from a remote DataWriter received for the first time by a local DataReader since the last time the status was read.

6.18.2.5 DDS_LongLong DDS_-DataReaderProtocolStatus::duplicate_sample_count

The number of samples from a remote DataWriter received, not for the first time, by a local DataReader.

Such samples can be redundant, out-of-order, etc. and are not stored in the reader's queue.

6.18.2.6 DDS_LongLong DDS_-DataReaderProtocolStatus::duplicate_-sample_count_change

The incremental change in the number of samples from a remote DataWriter received, not for the first time, by a local DataReader since the last time the status was read.

Such samples can be redundant, out-of-order, etc. and are not stored in the reader's queue.

6.18.2.7 DDS_LongLong DDS_DataReaderProtocolStatus::duplicate_sample_bytes

The number of bytes of samples from a remote DataWriter received, not for the first time, by a local DataReader.

Such samples can be redundant, out-of-order, etc. and are not stored in the reader's queue.

6.18.2.8 DDS_LongLong DDS_DataReaderProtocolStatus::duplicate_sample_bytes_change

The incremental change in the number of bytes of samples from a remote DataWriter received, not for the first time, by a local DataReader since the last time the status was read.

Such samples can be redundant, out-of-order, etc. and are not stored in the reader's queue.

6.18.2.9 DDS_LongLong DDS_DataReaderProtocolStatus::filtered_sample_count

The number of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter.

6.18.2.10 DDS_LongLong DDS_DataReaderProtocolStatus::filtered_sample_count_change

The incremental change in the number of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter since the last time the status was read.

6.18.2.11 DDS_LongLong DDS_DataReaderProtocolStatus::filtered_sample_bytes

The number of bytes of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter.

6.18.2.12 DDS_LongLong DDS_DataReaderProtocolStatus::filtered_sample_bytes_change

The incremental change in the number of bytes of user samples filtered by the local DataReader due to Content-Filtered Topics or Time-Based Filter since the last time the status was read.

6.18.2.13 DDS_LongLong DDS_DataReaderProtocolStatus::received_heartbeat_count

The number of Heartbeats from a remote DataWriter received by a local DataReader.

6.18.2.14 DDS_LongLong DDS_DataReaderProtocolStatus::received_heartbeat_count_change

The incremental change in the number of Heartbeats from a remote DataWriter received by a local DataReader since the last time the status was read.

6.18.2.15 DDS_LongLong DDS_DataReaderProtocolStatus::received_heartbeat_bytes

The number of bytes of Heartbeats from a remote DataWriter received by a local DataReader.

6.18.2.16 DDS_LongLong DDS_DataReaderProtocolStatus::received_heartbeat_bytes_change

The incremental change in the number of bytes of Heartbeats from a remote DataWriter received by a local DataReader since the last time the status was read.

6.18.2.17 DDS_LongLong DDS_DataReaderProtocolStatus::sent_ack_count

The number of ACKs sent from a local DataReader to a matching remote DataWriter.

6.18.2.18 DDS_LongLong DDS_DataReaderProtocolStatus::sent_ack_count_change

The incremental change in the number of ACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

6.18.2.19 DDS_LongLong DDS_DataReaderProtocolStatus::sent_ack_bytes

The number of bytes of ACKs sent from a local DataReader to a matching remote DataWriter.

6.18.2.20 DDS_LongLong DDS_DataReaderProtocolStatus::sent_ack_bytes_change

The incremental change in the number of bytes of ACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

6.18.2.21 DDS_LongLong DDS_DataReaderProtocolStatus::sent_nack_count

The number of NACKs sent from a local DataReader to a matching remote DataWriter.

6.18.2.22 DDS_LongLong DDS_DataReaderProtocolStatus::sent_nack_count_change

The incremental change in the number of NACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

6.18.2.23 DDS_LongLong DDS_DataReaderProtocolStatus::sent_nack_bytes

The number of bytes of NACKs sent from a local DataReader to a matching remote DataWriter.

6.18.2.24 DDS_LongLong DDS_DataReaderProtocolStatus::sent_nack_bytes_change

The incremental change in the number of bytes of NACKs sent from a local DataReader to a matching remote DataWriter since the last time the status was read.

6.18.2.25 DDS_LongLong DDS_DataReaderProtocolStatus::received_gap_count

The number of GAPs received from remote DataWriter to this DataReader.

6.18.2.26 DDS_LongLong DDS_-DataReaderProtocolStatus::received_gap_count_change

The incremental change in the number of GAPs received from remote DataWriter to this DataReader since the last time the status was read.

6.18.2.27 DDS_LongLong DDS_-DataReaderProtocolStatus::received_gap_bytes

The number of bytes of GAPs received from remote DataWriter to this DataReader.

6.18.2.28 DDS_LongLong DDS_-DataReaderProtocolStatus::received_gap_bytes_change

The incremental change in the number of bytes of GAPs received from remote DataWriter to this DataReader since the last time the status was read.

6.18.2.29 DDS_LongLong DDS_-DataReaderProtocolStatus::rejected_sample_count

The number of times a sample is rejected due to exceptions in the receive path.

6.18.2.30 DDS_LongLong DDS_-DataReaderProtocolStatus::rejected_-sample_count_change

The incremental change in the number of times a sample is rejected due to exceptions in the receive path since the last time the status was read.

6.18.2.31 struct DDS_SequenceNumber_t DDS_DataReaderProtocolStatus::first_available_sample_-sequence_number [read]

Sequence number of the first available sample in a matched Datawriters reliability queue.

Applicable only for reliable DataReaders, and when retrieving matched DataWriter statuses.

Updated upon receiving Heartbeat submessages from a matched reliable DataWriter.

6.18.2.32 struct DDS_SequenceNumber_t DDS_DataReaderProtocolStatus::last_available_sample_sequence_number [read]

Sequence number of the last available sample in a matched Datawriter's reliability queue.

Applicable only for reliable DataReaders, and when retrieving matched DataWriter statuses.

Updated upon receiving Heartbeat submessages from a matched reliable DataWriter.

6.18.2.33 struct DDS_SequenceNumber_t DDS_DataReaderProtocolStatus::last_committed_sample_sequence_number [read]

Sequence number of the newest sample received from the matched DataWriter committed to the DataReader's queue.

Applicable only when retrieving matched DataWriter statuses.

For best-effort DataReaders, this is the sequence number of the latest sample received.

For reliable DataReaders, this is the sequence number of the latest sample that is available to be read or taken from the DataReader's queue.

6.18.2.34 DDS_Long DDS_DataReaderProtocolStatus::uncommitted_sample_count

Number of received samples that are not yet available to be read or taken, due to being received out of order.

Applicable only when retrieving matched DataWriter statuses.

6.19 DDS_DataReaderQos Struct Reference

QoS policies supported by a **DDSDataReader** (p. 1087) entity.

Public Attributes

- struct **DDS_DurabilityQosPolicy durability**
  *Durability policy, **DURABILITY** (p. 348).*
- struct **DDS_DeadlineQosPolicy deadline**
  *Deadline policy, **DEADLINE** (p. 353).*
- struct **DDS_LatencyBudgetQosPolicy latency_budget**
  *Latency budget policy, **LATENCY_BUDGET** (p. 354).*
- struct **DDS_LivelinessQosPolicy liveliness**
  *Liveliness policy, **LIVELINESS** (p. 358).*
- struct **DDS_ReliabilityQosPolicy reliability**
  *Reliability policy, **RELIABILITY** (p. 362).*
- struct **DDS_DestinationOrderQosPolicy destination_order**
  *Destination order policy, **DESTINATION_ORDER** (p. 365).*
- struct **DDS_HistoryQosPolicy history**
  *History policy, **HISTORY** (p. 367).*
- struct **DDS_ResourceLimitsQosPolicy resource_limits**
  *Resource limits policy, **RESOURCE_LIMITS** (p. 371).*
- struct **DDS_UserDataQosPolicy user_data**
  *User data policy, **USER_DATA** (p. 345).*
- struct **DDS_OwnershipQosPolicy ownership**
  *Ownership policy, **OWNERSHIP** (p. 355).*
- struct **DDS_TimeBasedFilterQosPolicy time_based_filter**
  *Time-based filter policy, **TIME_BASED_FILTER** (p. 360).*
- struct **DDS_ReaderDataLifecycleQosPolicy reader_data_lifecycle**
  *Reader data lifecycle policy, **READER_DATA_LIFECYCLE** (p. 376).*

^ struct **DDS_TypeConsistencyEnforcementQosPolicy type_consistency**

Type consistency enforcement policy, ***TYPE_CONSISTENCY_ENFORCEMENT*** *(p. 432).*

^ struct **DDS_DataReaderResourceLimitsQosPolicy reader_resource_limits**

<<**eXtension**>> (p. 199) ***DDSDataReader*** *(p. 1087) resource limits policy,* ***DATA_READER_RESOURCE_LIMITS*** *(p. 407). This policy is an extension to the DDS standard.*

^ struct **DDS_DataReaderProtocolQosPolicy protocol**

<<**eXtension**>> (p. 199) ***DDSDataReader*** *(p. 1087) protocol policy,* ***DATA_READER_PROTOCOL*** *(p. 413)*

^ struct **DDS_TransportSelectionQosPolicy transport_selection**

<<**eXtension**>> (p. 199) *Transport selection policy,* ***TRANSPORT_SELECTION*** *(p. 382).*

^ struct **DDS_TransportUnicastQosPolicy unicast**

<<**eXtension**>> (p. 199) *Unicast transport policy,* ***TRANSPORT_UNICAST*** *(p. 383).*

^ struct **DDS_TransportMulticastQosPolicy multicast**

<<**eXtension**>> (p. 199) *Multicast transport policy,* ***TRANSPORT_MULTICAST*** *(p. 384).*

^ struct **DDS_PropertyQosPolicy property**

<<**eXtension**>> (p. 199) *Property policy,* ***PROPERTY*** *(p. 436).*

^ struct **DDS_AvailabilityQosPolicy availability**

<<**eXtension**>> (p. 199) *Availability policy,* ***AVAILABILITY*** *(p. 442).*

^ struct **DDS_EntityNameQosPolicy subscription_name**

<<**eXtension**>> (p. 199) *EntityName policy,* ***ENTITY_NAME*** *(p. 445).*

^ struct **DDS_TypeSupportQosPolicy type_support**

<<**eXtension**>> (p. 199) *type support data,* ***TYPESUPPORT*** *(p. 429).*

6.19.1 Detailed Description

QoS policies supported by a **DDSDataReader** (p. 1087) entity.

You must set certain members in a consistent manner:

DDS_DataReaderQos::deadline.period >= DDS_DataReaderQos::time_based_filter.minimum_separation

DDS_DataReaderQos::history.depth <= DDS_DataReaderQos::resource_limits.max_samples_per_instance

DDS_DataReaderQos::resource_limits.max_samples_per_instance <= DDS_DataReaderQos::resource_limits.max_samples DDS_DataReaderQos::resource_limits.initial_samples <= DDS_DataReaderQos::resource_limits.max_samples

DDS_DataReaderQos::resource_limits.initial_instances <= DDS_DataReaderQos::resource_limits.max_instances

DDS_DataReaderQos::reader_resource_limits.initial_remote_writers_per_instance <= DDS_DataReaderQos::reader_resource_limits.max_remote_writers_per_instance

DDS_DataReaderQos::reader_resource_limits.initial_infos <= DDS_DataReaderQos::reader_resource_limits.max_infos

DDS_DataReaderQos::reader_resource_limits.max_remote_writers_per_instance <= DDS_DataReaderQos::reader_resource_limits.max_remote_writers

DDS_DataReaderQos::reader_resource_limits.max_samples_per_remote_writer <= DDS_DataReaderQos::resource_limits.max_samples

length of DDS_DataReaderQos::user_data.value <= DDS_DomainParticipantQos::resource_limits.reader_user_data_max_length

If any of the above are not true, **DDSDataReader::set_qos** (p. 1102) and **DDSDataReader::set_qos_with_profile** (p. 1103) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.19.2 Member Data Documentation

6.19.2.1 struct DDS_DurabilityQosPolicy DDS_DataReaderQos::durability [read]

Durability policy, **DURABILITY** (p. 348).

6.19.2.2 struct DDS_DeadlineQosPolicy DDS_DataReaderQos::deadline [read]

Deadline policy, **DEADLINE** (p. 353).

6.19.2.3 struct DDS_LatencyBudgetQosPolicy DDS_DataReaderQos::latency_budget [read]

Latency budget policy, **LATENCY_BUDGET** (p. 354).

6.19.2.4 struct DDS_LivelinessQosPolicy DDS_DataReaderQos::liveliness [read]

Liveliness policy, **LIVELINESS** (p. 358).

6.19.2.5 struct DDS_ReliabilityQosPolicy DDS_DataReaderQos::reliability [read]

Reliability policy, **RELIABILITY** (p. 362).

6.19.2.6 struct DDS_DestinationOrderQosPolicy DDS_DataReaderQos::destination_order [read]

Destination order policy, **DESTINATION_ORDER** (p. 365).

6.19.2.7 struct DDS_HistoryQosPolicy DDS_DataReaderQos::history [read]

History policy, **HISTORY** (p. 367).

6.19.2.8 struct DDS_ResourceLimitsQosPolicy DDS_DataReaderQos::resource_limits [read]

Resource limits policy, **RESOURCE_LIMITS** (p. 371).

6.19.2.9 struct DDS_UserDataQosPolicy DDS_DataReaderQos::user_data [read]

User data policy, **USER_DATA** (p. 345).

6.19.2.10 struct DDS_OwnershipQosPolicy DDS_DataReaderQos::ownership [read]

Ownership policy, **OWNERSHIP** (p. 355).

6.19.2.11 struct DDS_TimeBasedFilterQosPolicy DDS_DataReaderQos::time_based_filter [read]

Time-based filter policy, **TIME_BASED_FILTER** (p. 360).

6.19.2.12 struct DDS_ReaderDataLifecycleQosPolicy DDS_DataReaderQos::reader_data_lifecycle [read]

Reader data lifecycle policy, **READER_DATA_LIFECYCLE** (p. 376).

6.19.2.13 struct DDS_TypeConsistencyEnforcementQosPolicy DDS_DataReaderQos::type_consistency [read]

Type consistency enforcement policy, **TYPE_CONSISTENCY_ENFORCEMENT** (p. 432).

6.19.2.14 struct DDS_DataReaderResourceLimitsQosPolicy DDS_DataReaderQos::reader_resource_limits [read]

<<*eXtension*>> *(p. 199)* **DDSDataReader** (p. 1087) resource limits policy, **DATA_READER_RESOURCE_LIMITS** (p. 407). This policy is an extension to the DDS standard.

6.19.2.15 struct DDS_DataReaderProtocolQosPolicy DDS_DataReaderQos::protocol [read]

<<*eXtension*>> *(p. 199)* **DDSDataReader** (p. 1087) protocol policy, **DATA_READER_PROTOCOL** (p. 413)

6.19.2.16 struct DDS_TransportSelectionQosPolicy DDS_DataReaderQos::transport_selection [read]

<<*eXtension*>> *(p. 199)* Transport selection policy, **TRANSPORT_SELECTION** (p. 382).

Specifies the transports available for use by the **DDSDataReader** (p. 1087).

6.19.2.17 struct DDS_TransportUnicastQosPolicy DDS_DataReaderQos::unicast [read]

<<*eXtension*>> *(p. 199)* Unicast transport policy, **TRANSPORT_UNICAST** (p. 383).

Specifies the unicast transport interfaces and ports on which **messages** can be received.

The unicast interfaces are used to receive messages from **DDSDataWriter** (p. 1113) entities in the domain.

6.19.2.18 struct DDS_TransportMulticastQosPolicy DDS_DataReaderQos::multicast [read]

<<*eXtension*>> *(p. 199)* Multicast transport policy, **TRANSPORT_MULTICAST** (p. 384).

Specifies the multicast group addresses and ports on which **messages** can be received.

The multicast addresses are used to receive messages from **DDSDataWriter** (p. 1113) entities in the domain.

6.19.2.19 struct DDS_PropertyQosPolicy DDS_DataReaderQos::property [read]

<<*eXtension*>> *(p. 199)* Property policy, **PROPERTY** (p. 436).

6.19.2.20 struct DDS_AvailabilityQosPolicy DDS_DataReaderQos::availability [read]

<<*eXtension*>> *(p. 199)* Availability policy, **AVAILABILITY** (p. 442).

6.19.2.21 struct DDS_EntityNameQosPolicy DDS_DataReaderQos::subscription_name [read]

<<*eXtension*>> *(p. 199)* EntityName policy, **ENTITY_NAME** (p. 445).

6.19.2.22 struct DDS_TypeSupportQosPolicy DDS_DataReaderQos::type_support [read]

<<*eXtension*>> *(p. 199)* type support data, **TYPESUPPORT** (p. 429).

Optional value that is passed to a type plugin's on_endpoint_attached and deserialization functions.

6.20 DDS_DataReaderResourceLimitsQosPolicy Struct Reference

Various settings that configure how a **DDSDataReader** (p. 1087) allocates and uses physical memory for internal resources.

Public Attributes

- **DDS_Long max_remote_writers**

  *The maximum number of remote writers from which a **DDSDataReader** (p. 1087) may read, including all instances.*

- **DDS_Long max_remote_writers_per_instance**

  *The maximum number of remote writers from which a **DDSDataReader** (p. 1087) may read a single instance.*

- **DDS_Long max_samples_per_remote_writer**

  *The maximum number of out-of-order samples from a given remote **DDSDataWriter** (p. 1113) that a **DDSDataReader** (p. 1087) may store when maintaining a reliable connection to the **DDSDataWriter** (p. 1113).*

- **DDS_Long max_infos**

  *The maximum number of info units that a **DDSDataReader** (p. 1087) can use to store **DDS_SampleInfo** (p. 912).*

- **DDS_Long initial_remote_writers**

  *The initial number of remote writers from which a **DDSDataReader** (p. 1087) may read, including all instances.*

- **DDS_Long initial_remote_writers_per_instance**

  *The initial number of remote writers from which a **DDSDataReader** (p. 1087) may read a single instance.*

- **DDS_Long initial_infos**

  *The initial number of info units that a **DDSDataReader** (p. 1087) can have, which are used to store **DDS_SampleInfo** (p. 912).*

- **DDS_Long initial_outstanding_reads**

  *The initial number of outstanding calls to read/take (or one of their variants) on the same **DDSDataReader** (p. 1087) for which memory has not been returned by calling **FooDataReader::return_loan** (p. 1471).*

- **DDS_Long max_outstanding_reads**

The maximum number of outstanding read/take calls (or one of their variants) on the same ***DDSDataReader*** *(*p. 1087*) for which memory has not been returned by calling* ***FooDataReader::return_loan*** *(*p. 1471*).*

ˆ DDS_Long max_samples_per_read

The maximum number of data samples that the application can receive from the middleware in a single call to ***FooDataReader::read*** *(*p. 1447*) or* ***FooDataReader::take*** *(*p. 1448*). If more data exists in the middleware, the application will need to issue multiple read/take calls.*

ˆ DDS_Boolean disable_fragmentation_support

Determines whether the ***DDSDataReader*** *(*p. 1087*) can receive fragmented samples.*

ˆ DDS_Long max_fragmented_samples

The maximum number of samples for which the ***DDSDataReader*** *(*p. 1087*) may store fragments at a given point in time.*

ˆ DDS_Long initial_fragmented_samples

The initial number of samples for which a ***DDSDataReader*** *(*p. 1087*) may store fragments.*

ˆ DDS_Long max_fragmented_samples_per_remote_writer

The maximum number of samples per remote writer for which a ***DDSDataReader*** *(*p. 1087*) may store fragments.*

ˆ DDS_Long max_fragments_per_sample

Maximum number of fragments for a single sample.

ˆ DDS_Boolean dynamically_allocate_fragmented_samples

Determines whether the ***DDSDataReader*** *(*p. 1087*) pre-allocates storage for storing fragmented samples.*

ˆ DDS_Long max_total_instances

Maximum number of instances for which a DataReader will keep state.

ˆ DDS_Long max_remote_virtual_writers

The maximum number of remote virtual writers from which a ***DDSDataReader*** *(*p. 1087*) may read, including all instances.*

ˆ DDS_Long initial_remote_virtual_writers

The initial number of remote virtual writers from which a ***DDSDataReader*** *(*p. 1087*) may read, including all instances.*

- **DDS_Long max_remote_virtual_writers_per_instance**

  The maximum number of virtual remote writers that can be associated with an instance.

- **DDS_Long initial_remote_virtual_writers_per_instance**

  The initial number of virtual remote writers per instance.

- **DDS_Long max_remote_writers_per_sample**

  The maximum number of remote writers allowed to write the same sample.

- **DDS_Long max_query_condition_filters**

  The maximum number of query condition filters a reader is allowed.

- **DDS_Long max_app_ack_response_length**

  Maximum length of application-level acknowledgment response data.

6.20.1 Detailed Description

Various settings that configure how a **DDSDataReader** (p. 1087) allocates and uses physical memory for internal resources.

DataReaders must allocate internal structures to handle the maximum number of DataWriters that may connect to it, whether or not a **DDSDataReader** (p. 1087) handles data fragmentation and how many data fragments that it may handle (for data samples larger than the MTU of the underlying network transport), how many simultaneous outstanding loans of internal memory holding data samples can be provided to user code, as well as others.

Most of these internal structures start at an initial size and, by default, will grow as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want to bound the amount of memory that can be used by a **DDSDataReader** (p. 1087). By setting the initial size to the maximum size, you will prevent RTI Connext from dynamically allocating any memory after the creation of the **DDSDataReader** (p. 1087).

This QoS policy is an extension to the DDS standard.

Entity:

DDSDataReader (p. 1087)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.20.2 Member Data Documentation

6.20.2.1 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_remote_writers

The maximum number of remote writers from which a **DDSDataReader** (p. 1087) may read, including all instances.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_remote_writers, >= max_remote_writers_per_instance

For unkeyed types, this value has to be equal to max_remote_writers_per_instance if max_remote_writers_per_instance is not equal to **DDS_LENGTH_UNLIMITED** (p. 371).

Note: For efficiency, set `max_remote_writers` >= **DDS_DataReaderResourceLimitsQosPolicy::max_remote_writers_per_instance** (p. 524).

6.20.2.2 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_remote_writers_per_instance

The maximum number of remote writers from which a **DDSDataReader** (p. 1087) may read a single instance.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1024] or **DDS_LENGTH_UNLIMITED** (p. 371), <= max_remote_writers or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_remote_writers_per_instance

For unkeyed types, this value has to be equal to max_remote_writers if it is not **DDS_LENGTH_UNLIMITED** (p. 371).

Note: For efficiency, set `max_remote_writers_per_instance` <= **DDS_DataReaderResourceLimitsQosPolicy::max_remote_writers** (p. 524)

6.20.2.3 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_samples_per_remote_writer

The maximum number of out-of-order samples from a given remote **DDSDataWriter** (p. 1113) that a **DDSDataReader** (p. 1087) may store when maintaining a reliable connection to the **DDSDataWriter** (p. 1113).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 100 million] or **DDS_LENGTH_UNLIMITED** (p. 371), <= **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881)

6.20.2.4 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_infos

The maximum number of info units that a **DDSDataReader** (p. 1087) can use to store **DDS_SampleInfo** (p. 912).

When read/take is called on a DataReader, the DataReader passes a sequence of data samples and an associated sample info sequence. The sample info sequence contains additional information for each data sample.

max_infos determines the resources allocated for storing sample info. This memory is loaned to the application when passing a sample info sequence.

Note that sample info is a snapshot, generated when read/take is called.

max_infos should not be less than max_samples.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_infos

6.20.2.5 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::initial_remote_writers

The initial number of remote writers from which a **DDSDataReader** (p. 1087) may read, including all instances.

[default] 2

[range] [1, 1 million], <= max_remote_writers

For unkeyed types this value has to be equal to initial_remote_writers_per_instance.

Note: For efficiency, set initial_remote_writers >= **DDS_DataReaderResourceLimitsQosPolicy::initial_remote_writers_per_instance** (p. 525).

6.20.2.6 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::initial_remote_writers_per_instance

The initial number of remote writers from which a **DDSDataReader** (p. 1087) may read a single instance.

[default] 2

[range] [1,1024], <= max_remote_writers_per_instance

For unkeyed types this value has to be equal to initial_remote_writers.

Note: For efficiency, set initial_remote_writers_per_instance <= **DDS_DataReaderResourceLimitsQosPolicy::initial_remote_writers** (p. 525).

6.20.2.7 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::initial_infos

The initial number of info units that a **DDSDataReader** (p. 1087) can have, which are used to store **DDS_SampleInfo** (p. 912).

[default] 32

[range] [1,1 million], <= max_infos

6.20.2.8 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::initial_outstanding_reads

The initial number of outstanding calls to read/take (or one of their variants) on the same **DDSDataReader** (p. 1087) for which memory has not been returned by calling **FooDataReader::return_loan** (p. 1471).

[default] 2

[range] [1, 65536], <= max_outstanding_reads

6.20.2.9 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_outstanding_reads

The maximum number of outstanding read/take calls (or one of their variants) on the same **DDSDataReader** (p. 1087) for which memory has not been returned by calling **FooDataReader::return_loan** (p. 1471).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 65536] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_outstanding_reads

6.20.2.10 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_samples_per_read

The maximum number of data samples that the application can receive from the middleware in a single call to **FooDataReader::read** (p. 1447) or **FooDataReader::take** (p. 1448). If more data exists in the middleware, the application will need to issue multiple read/take calls.

When reading data using listeners, the expected number of samples available for delivery in a single `take` call is typically small: usually just one, in the case of unbatched data, or the number of samples in a single batch, in the case of batched data. (See **DDS_BatchQosPolicy** (p. 476) for more information about this feature.) When polling for data or using a **DDSWaitSet** (p. 1433), however, multiple samples (or batches) could be retrieved at once, depending on the data rate.

A larger value for this parameter makes the API simpler to use at the expense of some additional memory consumption.

[default] 1024

[range] [1,65536]

6.20.2.11 DDS_Boolean DDS_DataReaderResourceLimitsQosPolicy::disable_fragmentation_support

Determines whether the **DDSDataReader** (p. 1087) can receive fragmented samples.

When fragmentation support is not needed, disabling fragmentation support will save some memory resources.

[default] **DDS_BOOLEAN_FALSE** (p. 299)

6.20.2.12 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_fragmented_samples

The maximum number of samples for which the **DDSDataReader** (p. 1087) may store fragments at a given point in time.

At any given time, a **DDSDataReader** (p. 1087) may store fragments for up to `max_fragmented_samples` samples while waiting for the remaining fragments. These samples need not have consecutive sequence numbers and may have been sent by different **DDSDataWriter** (p. 1113) instances.

Once all fragments of a sample have been received, the sample is treated as a

regular sample and becomes subject to standard QoS settings such as **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881).

The middleware will drop fragments if the `max_fragmented_samples` limit has been reached. For best-effort communication, the middleware will accept a fragment for a new sample, but drop the oldest fragmented sample from the same remote writer. For reliable communication, the middleware will drop fragments for any new samples until all fragments for at least one older sample from that writer have been received.

Only applies if **DDS_DataReaderResourceLimitsQosPolicy::disable_fragmentation_support** (p. 527) is **DDS_BOOLEAN_FALSE** (p. 299).

[default] 1024

[range] [1, 1 million]

6.20.2.13 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::initial_fragmented_samples

The initial number of samples for which a **DDSDataReader** (p. 1087) may store fragments.

Only applies if **DDS_DataReaderResourceLimitsQosPolicy::disable_fragmentation_support** (p. 527) is **DDS_BOOLEAN_FALSE** (p. 299).

[default] 4

[range] [1,1024], <= max_fragmented_samples

6.20.2.14 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_fragmented_samples_per_remote_writer

The maximum number of samples per remote writer for which a **DDSDataReader** (p. 1087) may store fragments.

Logical limit so a single remote writer cannot consume all available resources.

Only applies if **DDS_DataReaderResourceLimitsQosPolicy::disable_fragmentation_support** (p. 527) is **DDS_BOOLEAN_FALSE** (p. 299).

[default] 256

[range] [1, 1 million], <= max_fragmented_samples

6.20.2.15 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_fragments_per_sample

Maximum number of fragments for a single sample.

Only applies if **DDS_DataReaderResourceLimitsQosPolicy::disable_fragmentation_support** (p. 527) is **DDS_BOOLEAN_FALSE** (p. 299).

[default] 512

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371)

6.20.2.16 DDS_Boolean DDS_DataReaderResourceLimitsQosPolicy::dynamically_allocate_fragmented_samples

Determines whether the **DDSDataReader** (p. 1087) pre-allocates storage for storing fragmented samples.

By default, the middleware will allocate memory upfront for storing fragments for up to **DDS_DataReaderResourceLimitsQosPolicy::initial_fragmented_samples** (p. 528) samples. This memory may grow up to **DDS_DataReaderResourceLimitsQosPolicy::max_fragmented_samples** (p. 527) if needed.

If `dynamically_allocate_fragmented_samples` is set to **DDS_BOOLEAN_TRUE** (p. 298), the middleware does not allocate memory upfront, but instead allocates memory from the heap upon receiving the first fragment of a new sample. The amount of memory allocated equals the amount of memory needed to store all fragments in the sample. Once all fragments of a sample have been received, the sample is deserialized and stored in the regular receive queue. At that time, the dynamically allocated memory is freed again.

This QoS setting may be useful for large, but variable-sized data types where upfront memory allocation for multiple samples based on the maximum possible sample size may be expensive. The main disadvantage of not pre-allocating memory is that one can no longer guarantee the middleware will have sufficient resources at run-time.

Only applies if **DDS_DataReaderResourceLimitsQosPolicy::disable_fragmentation_support** (p. 527) is **DDS_BOOLEAN_FALSE** (p. 299).

[default] **DDS_BOOLEAN_FALSE** (p. 299)

6.20.2.17 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_total_instances

Maximum number of instances for which a DataReader will keep state.

The maximum number of instances actively managed by a DataReader is determined by **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882).

These instances have associated DataWriters or samples in the DataReader's queue and are visible to the user through operations such as **FooDataReader::take** (p. 1448), **FooDataReader::read** (p. 1447), and **FooDataReader::get_key_value** (p. 1472).

The features Durable Reader State, MultiChannel DataWriters and RTI Persistence Service require RTI Connext to keep some internal state even for instances without DataWriters or samples in the DataReader's queue. The additional state is used to filter duplicate samples that could be coming from different DataWriter channels or from multiple executions of RTI Persistence Service.

The total maximum number of instances that will be managed by the middleware, including instances without associated DataWriters or samples, is determined by max_total_instances.

When a new instance is received, RTI Connext will check the resource limit **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882). If the limit is exceeded, RTI Connext will drop the sample and report it as lost and rejected. If the limit is not exceeded, RTI Connext will check max_total_instances. If max_total_instances is exceeded, RTI Connext will replace an existing instance without DataWriters and samples with the new one. The application could receive duplicate samples for the replaced instance if it becomes alive again.

[default] **DDS_AUTO_MAX_TOTAL_INSTANCES** (p. 408)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371) or **DDS_AUTO_MAX_TOTAL_INSTANCES** (p. 408), >= **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882)

6.20.2.18 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_remote_virtual_writers

The maximum number of remote virtual writers from which a **DDSDataReader** (p. 1087) may read, including all instances.

When **DDS_PresentationQosPolicy::access_scope** (p. 826) is set to **DDS_GROUP_PRESENTATION_QOS** (p. 352), this value determines the maximum number of DataWriter groups that can be managed by the **DDSSubscriber** (p. 1390) containing this **DDSDataReader** (p. 1087).

Since the **DDSSubscriber** (p. 1390) may contain more than one **DDSDataReader** (p. 1087), only the setting of the first applies.

[default] DDS_LENGTH_UNLIMITED (p. 371)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_remote_virtual_writers, >= max_remote_virtual_writers_per_instance

6.20.2.19 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::initial_remote_virtual_writers

The initial number of remote virtual writers from which a **DDSDataReader** (p. 1087) may read, including all instances.

[default] 2

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), <= max_remote_virtual_writers

6.20.2.20 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_remote_virtual_writers_per_instance

The maximum number of virtual remote writers that can be associated with an instance.

[default] DDS_LENGTH_UNLIMITED (p. 371)

[range] [1, 1024] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_remote_virtual_writers_per_instance

For unkeyed types, this value is ignored.

The features of Durable Reader State and MultiChannel DataWriters, and RTI Persistence Service require RTI Connext to keep some internal state per virtual writer and instance that is used to filter duplicate samples. These duplicate samples could be coming from different DataWriter channels or from multiple executions of RTI Persistence Service.

Once an association between a remote virtual writer and an instance is established, it is permanent – it will not disappear even if the physical writer incarnating the virtual writer is destroyed.

If max_remote_virtual_writers_per_instance is exceeded for an instance, RTI Connext will not associate this instance with new virtual writers. Duplicates samples from these virtual writers will not be filtered on the reader.

If you are not using Durable Reader State, MultiChannel DataWriters or RTI Persistence Service in your system, you can set this property to 1 to optimize

resources.

6.20.2.21 DDS_Long DDS_-DataReaderResourceLimitsQosPolicy::initial_remote_-virtual_writers_per_instance

The initial number of virtual remote writers per instance.

[default] 2

[range] [1, 1024], <= max_remote_virtual_writers_per_instance

For unkeyed types, this value is ignored.

6.20.2.22 DDS_Long DDS_-DataReaderResourceLimitsQosPolicy::max_remote_-writers_per_sample

The maximum number of remote writers allowed to write the same sample.

One scenario in which two DataWriters may write the same sample is Persistence Service. The DataReader may receive the same sample coming from the original DataWriter and from a Persistence Service DataWriter. **[default]** 3

[range] [1, 1024]

6.20.2.23 DDS_Long DDS_-DataReaderResourceLimitsQosPolicy::max_query_-condition_filters

The maximum number of query condition filters a reader is allowed.

[default] 4

[range] [0, 32]

This value determines the maximum number of unique query condition content filters that a reader may create.

Each query condition content filter is comprised of both its `query_expression` and `query_parameters`. Two query conditions that have the same `query_-expression` will require unique query condition filters if their `query_paramters` differ. Query conditions that differ only in their state masks will share the same query condition filter.

6.20.2.24 DDS_Long DDS_DataReaderResourceLimitsQosPolicy::max_app_ack_response_length

Maximum length of application-level acknowledgment response data.

The maximum length of response data in an application-level acknowledgment.

When set to zero, no response data is sent with application-level acknowledgments.

[default] 0

[range] [0, 65536]

6.21 DDS_DataWriterCacheStatus Struct Reference

<<*eXtension*>> *(*p. *199)* The status of the writer's cache.

Public Attributes

- **DDS_LongLong sample_count_peak**

  Highest number of samples in the writer's queue over the lifetime of the writer.

- **DDS_LongLong sample_count**

  Number of samples in the writer's queue.

6.21.1 Detailed Description

<<*eXtension*>> *(*p. *199)* The status of the writer's cache.

Entity:

DDSDataWriter (p. 1113)

6.21.2 Member Data Documentation

6.21.2.1 DDS_LongLong DDS_DataWriterCacheStatus::sample_count_peak

Highest number of samples in the writer's queue over the lifetime of the writer.

6.21.2.2 DDS_LongLong DDS_DataWriterCacheStatus::sample_count

Number of samples in the writer's queue.

6.22 DDS_DataWriterProtocolQosPolicy Struct Reference

Protocol that applies only to **DDSDataWriter** (p. 1113) instances.

Public Attributes

- struct **DDS_GUID_t virtual_guid**

  The virtual GUID (Global Unique Identifier).

- **DDS_UnsignedLong rtps_object_id**

  The RTPS Object ID.

- **DDS_Boolean push_on_write**

  Whether to push sample out when write is called.

- **DDS_Boolean disable_positive_acks**

  Controls whether or not the writer expects positive acknowledgements from matching readers.

- **DDS_Boolean disable_inline_keyhash**

  Controls whether or not a keyhash is propagated on the wire with each sample.

- **DDS_Boolean serialize_key_with_dispose**

  Controls whether or not the serialized key is propagated on the wire with dispose samples.

- struct **DDS_RtpsReliableWriterProtocol_t rtps_reliable_writer**

  The reliable protocol defined in RTPS.

6.22.1 Detailed Description

Protocol that applies only to **DDSDataWriter** (p. 1113) instances.

DDS has a standard protocol for packet (user and meta data) exchange between applications using DDS for communications. This QoS policy and **DDS_DataWriterProtocolQosPolicy** (p. 535) give you control over configurable portions of the protocol, including the configuration of the reliable data delivery mechanism of the protocol on a per DataWriter or DataReader basis.

These configuration parameters control timing, timeouts, and give you the ability to tradeoff between speed of data loss detection and repair versus network and CPU bandwidth used to maintain reliability.

It is important to tune the reliability protocol (on a per **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) basis) to meet the requirements of the end-user application so that data can be sent between DataWriters and DataReaders in an efficient and optimal manner in the presence of data loss.

You can also use this QoS policy to control how RTI Connext responds to "slow" reliable DataReaders or ones that disconnect or are otherwise lost. See **DDS_ReliabilityQosPolicy** (p. 865) for more information on the per-DataReader/DataWriter reliability configuration. **DDS_HistoryQosPolicy** (p. 758) and **DDS_ResourceLimitsQosPolicy** (p. 879) also play an important role in the DDS reliable protocol.

This QoS policy is an extension to the DDS standard.

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.22.2 Member Data Documentation

6.22.2.1 struct DDS_GUID_t DDS_DataWriterProtocolQosPolicy::virtual_guid [read]

The virtual GUID (Global Unique Identifier).

The virtual GUID is used to uniquely identify different incarnations of the same **DDSDataWriter** (p. 1113).

RTI Connext uses the virtual GUID to associate a persisted writer history to a specific **DDSDataWriter** (p. 1113).

The RTI Connext Persistence Service uses the virtual GUID to send samples on behalf of the original **DDSDataWriter** (p. 1113).

[**default**] **DDS_GUID_AUTO** (p. 309)

6.22.2.2 DDS_UnsignedLong DDS_DataWriterProtocolQosPolicy::rtps_object_id

The RTPS Object ID.

This value is used to determine the RTPS object ID of a data writer according to the DDS-RTPS Interoperability Wire Protocol.

Only the last 3 bytes are used; the most significant byte is ignored.

If the default value is specified, RTI Connext will automatically assign the object ID based on a counter value (per participant) starting at 0x00800000. That value is incremented for each new data writer.

A rtps_object_id value in the interval [0x00800000,0x00ffffff] may collide with the automatic values assigned by RTI Connext. In those cases, the recomendation is not to use automatic object ID assigment.

[default] DDS_RTPS_AUTO_ID (p. **??**)

[range] [0,0x00ffffff]

6.22.2.3 DDS_Boolean DDS_DataWriterProtocolQosPolicy::push_on_write

Whether to push sample out when write is called.

If set to **DDS_BOOLEAN_TRUE** (p. 298) (the default), the writer will send a sample every time write is called. Otherwise, the sample is put into the queue waiting for a NACK from remote reader(s) to be sent out.

[default] DDS_BOOLEAN_TRUE (p. 298)

6.22.2.4 DDS_Boolean DDS_DataWriterProtocolQosPolicy::disable_positive_acks

Controls whether or not the writer expects positive acknowledgements from matching readers.

If set to **DDS_BOOLEAN_TRUE** (p. 298), the writer does not expect readers to send send positive acknowledgments to the writer. Consequently, instead of keeping a sample queued until all readers have positively acknowledged it, the writer will keep a sample for at least **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_min_sample_keep_duration** (p. 898), after which the sample is logically considered as positively acknowledged.

If set to **DDS_BOOLEAN_FALSE** (p. 299) (the default), the writer expects to receive positive acknowledgements from its acknowledging readers (**DDS_DataReaderProtocolQosPolicy::disable_positive_acks** (p. 503) = **DDS_BOOLEAN_FALSE** (p. 299)) and it applies the keep-duration to its non-acknowledging readers (**DDS_DataReaderProtocolQosPolicy::disable_positive_acks** (p. 503) = **DDS_BOOLEAN_TRUE** (p. 298)).

A writer with both acknowledging and non-acknowledging readers keeps a sample queued until acknowledgements have been received from all acknowledging readers and the keep-duration has elapsed for non-acknowledging readers.

[default] DDS_BOOLEAN_FALSE (p. 299)

6.22.2.5 DDS_Boolean DDS_DataWriterProtocolQosPolicy::disable_inline_keyhash

Controls whether or not a keyhash is propagated on the wire with each sample.

This field only applies to keyed writers.

With each key, RTI Connext associates an internal 16-byte representation, called a keyhash.

When this field is **DDS_BOOLEAN_FALSE** (p. 299), the keyhash is sent on the wire with every data instance.

When this field is **DDS_BOOLEAN_TRUE** (p. 298), the keyhash is not sent on the wire and the readers must compute the value using the received data.

If the *reader* is CPU bound, sending the keyhash on the wire may increase performance, because the reader does not have to get the keyhash from the data.

If the *writer* is CPU bound, sending the keyhash on the wire may decrease performance, because it requires more bandwidth (16 more bytes per sample).

Note: Setting disable_inline_keyhash to **DDS_BOOLEAN_TRUE** (p. 298) is not compatible with using RTI Real-Time Connect or RTI Recorder.

[default] DDS_BOOLEAN_FALSE (p. 299)

6.22.2.6 DDS_Boolean DDS_DataWriterProtocolQosPolicy::serialize_key_with_dispose

Controls whether or not the serialized key is propagated on the wire with dispose samples.

This field only applies to keyed writers.

We recommend setting this field to **DDS_BOOLEAN_TRUE** (p. 298) if there are DataReaders where **DDS_DataReaderProtocolQosPolicy::propagate_dispose_of_unregistered_instances** (p. 504) is also **DDS_BOOLEAN_TRUE** (p. 298).

Important: When this field is **DDS_BOOLEAN_TRUE** (p. 298), batching will not be compatible with RTI Connext 4.3e, 4.4b, or 4.4c. The **DDSDataReader** (p. 1087) entities will receive incorrect data and/or encounter deserialization errors.

[default] DDS_BOOLEAN_FALSE (p. 299)

6.22.2.7 struct DDS_RtpsReliableWriterProtocol_t DDS_DataWriterProtocolQosPolicy::rtps_reliable_writer [read]

The reliable protocol defined in RTPS.

[**default**] low_watermark 0;

high_watermark 1;

heartbeat_period 3.0 seconds;

fast_heartbeat_period 3.0 seconds;

late_joiner_heartbeat_period 3.0 seconds;

virtual_heartbeat_period **DDS_DURATION_AUTO** (p. 305);

samples_per_virtual_heartbeat **DDS_LENGTH_UNLIMITED** (p. 371);

max_heartbeat_retries 10;

inactivate_nonprogressing_readers **DDS_BOOLEAN_FALSE** (p. 299);

heartbeats_per_max_samples 8;

min_nack_response_delay 0.0 seconds;

max_nack_response_delay 0.2 seconds;

max_bytes_per_nack_response 131072

6.23 DDS_DataWriterProtocolStatus Struct Reference

*<<**eXtension**>>* *(p. 199)* The status of a writer's internal protocol related metrics, like the number of samples pushed, pulled, filtered; and status of wire protocol traffic.

Public Attributes

- **DDS_LongLong pushed_sample_count**

  The number of user samples pushed on write from a local DataWriter to a matching remote DataReader.

- **DDS_LongLong pushed_sample_count_change**

  The incremental change in the number of user samples pushed on write from a local DataWriter to a matching remote DataReader since the last time the status was read.

- **DDS_LongLong pushed_sample_bytes**

  The number of bytes of user samples pushed on write from a local DataWriter to a matching remote DataReader.

- **DDS_LongLong pushed_sample_bytes_change**

  The incremental change in the number of bytes of user samples pushed on write from a local DataWriter to a matching remote DataReader since the last time the status was read.

- **DDS_LongLong filtered_sample_count**

  The number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics.

- **DDS_LongLong filtered_sample_count_change**

  The incremental change in the number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics since the last time the status was read.

- **DDS_LongLong filtered_sample_bytes**

  The number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics.

- **DDS_LongLong filtered_sample_bytes_change**

  The incremental change in the number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics since the last time the status was read.

ˆ **DDS_LongLong sent_heartbeat_count**

The number of Heartbeats sent between a local DataWriter and matching remote DataReader.

ˆ **DDS_LongLong sent_heartbeat_count_change**

The incremental change in the number of Heartbeats sent between a local DataWriter and matching remote DataReader since the last time the status was read.

ˆ **DDS_LongLong sent_heartbeat_bytes**

The number of bytes of Heartbeats sent between a local DataWriter and matching remote DataReader.

ˆ **DDS_LongLong sent_heartbeat_bytes_change**

The incremental change in the number of bytes of Heartbeats sent between a local DataWriter and matching remote DataReader since the last time the status was read.

ˆ **DDS_LongLong pulled_sample_count**

The number of user samples pulled from local DataWriter by matching DataReaders.

ˆ **DDS_LongLong pulled_sample_count_change**

The incremental change in the number of user samples pulled from local DataWriter by matching DataReaders since the last time the status was read.

ˆ **DDS_LongLong pulled_sample_bytes**

The number of bytes of user samples pulled from local DataWriter by matching DataReaders.

ˆ **DDS_LongLong pulled_sample_bytes_change**

The incremental change in the number of bytes of user samples pulled from local DataWriter by matching DataReaders since the last time the status was read.

ˆ **DDS_LongLong received_ack_count**

The number of ACKs from a remote DataReader received by a local DataWriter.

ˆ **DDS_LongLong received_ack_count_change**

The incremental change in the number of ACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

^ **DDS_LongLong received_ack_bytes**

The number of bytes of ACKs from a remote DataReader received by a local DataWriter.

^ **DDS_LongLong received_ack_bytes_change**

The incremental change in the number of bytes of ACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

^ **DDS_LongLong received_nack_count**

The number of NACKs from a remote DataReader received by a local DataWriter.

^ **DDS_LongLong received_nack_count_change**

The incremental change in the number of NACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

^ **DDS_LongLong received_nack_bytes**

The number of bytes of NACKs from a remote DataReader received by a local DataWriter.

^ **DDS_LongLong received_nack_bytes_change**

The incremental change in the number of bytes of NACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

^ **DDS_LongLong sent_gap_count**

The number of GAPs sent from local DataWriter to matching remote DataReaders.

^ **DDS_LongLong sent_gap_count_change**

The incremental change in the number of GAPs sent from local DataWriter to matching remote DataReaders since the last time the status was read.

^ **DDS_LongLong sent_gap_bytes**

The number of bytes of GAPs sent from local DataWriter to matching remote DataReaders.

^ **DDS_LongLong sent_gap_bytes_change**

The incremental change in the number of bytes of GAPs sent from local DataWriter to matching remote DataReaders since the last time the status was read.

^ **DDS_LongLong rejected_sample_count**

The number of times a sample is rejected due to exceptions in the send path.

ˆ **DDS_LongLong rejected_sample_count_change**

The incremental change in the number of times a sample is rejected due to exceptions in the send path since the last time the status was read.

ˆ **DDS_Long send_window_size**

Current maximum number of outstanding samples allowed in the DataWriter's queue.

ˆ struct **DDS_SequenceNumber_t first_available_sample_sequence_number**

The sequence number of the first available sample currently queued in the local DataWriter.

ˆ struct **DDS_SequenceNumber_t last_available_sample_sequence_number**

The sequence number of the last available sample currently queued in the local DataWriter.

ˆ struct **DDS_SequenceNumber_t first_unacknowledged_sample_sequence_number**

The sequence number of the first unacknowledged sample currently queued in the local DataWriter.

ˆ struct **DDS_SequenceNumber_t first_available_sample_virtual_sequence_number**

The virtual sequence number of the first available sample currently queued in the local DataWriter.

ˆ struct **DDS_SequenceNumber_t last_available_sample_virtual_sequence_number**

The virtual sequence number of the last available sample currently queued in the local DataWriter.

ˆ struct **DDS_SequenceNumber_t first_unacknowledged_sample_virtual_sequence_number**

The virtual sequence number of the first unacknowledged sample currently queued in the local DataWriter.

ˆ **DDS_InstanceHandle_t first_unacknowledged_sample_subscription_handle**

The handle of a remote DataReader that has not acknowledged the first unacknowledged sample of the local DataWriter.

^ struct **DDS_SequenceNumber_t first_unelapsed_keep_duration_sample_sequence_number**

The sequence number of the first sample whose keep duration has not yet elapsed.

6.23.1 Detailed Description

<<***eXtension***>> *(p. 199)* The status of a writer's internal protocol related metrics, like the number of samples pushed, pulled, filtered; and status of wire protocol traffic.

Entity:

DDSDataWriter (p. 1113)

6.23.2 Member Data Documentation

6.23.2.1 DDS_LongLong DDS_DataWriterProtocolStatus::pushed_sample_count

The number of user samples pushed on write from a local DataWriter to a matching remote DataReader.

Counts protocol (RTPS) messages pushed by a DataWriter when writing, unregistering, and disposing. The count is the number of sends done internally, and it may be greater than the number of user writes.

For large data, counts whole samples, not fragments.

6.23.2.2 DDS_LongLong DDS_DataWriterProtocolStatus::pushed_sample_count_change

The incremental change in the number of user samples pushed on write from a local DataWriter to a matching remote DataReader since the last time the status was read.

Counts protocol (RTPS) messages pushed by a DataWriter when writing, unregistering, and disposing.

For large data, counts whole samples, not fragments.

6.23.2.3 DDS_LongLong DDS_DataWriterProtocolStatus::pushed_sample_bytes

The number of bytes of user samples pushed on write from a local DataWriter to a matching remote DataReader.

Counts bytes of protocol (RTPS) messages pushed by a DataWriter when writing, unregistering, and disposing. The count of bytes corresponds to the number of sends done internally, and it may be greater than the number of user writes.

For large data, counts bytes of whole samples, not fragments.

6.23.2.4 DDS_LongLong DDS_DataWriterProtocolStatus::pushed_sample_bytes_change

The incremental change in the number of bytes of user samples pushed on write from a local DataWriter to a matching remote DataReader since the last time the status was read.

Counts bytes of protocol (RTPS) messages pushed by a DataWriter when writing, unregistering, and disposing.

For large data, counts bytes of whole samples, not fragments.

6.23.2.5 DDS_LongLong DDS_DataWriterProtocolStatus::filtered_sample_count

The number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics.

6.23.2.6 DDS_LongLong DDS_DataWriterProtocolStatus::filtered_sample_count_change

The incremental change in the number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics since the last time the status was read.

6.23.2.7 DDS_LongLong DDS_DataWriterProtocolStatus::filtered_sample_bytes

The number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics.

6.23.2.8 DDS_LongLong DDS_DataWriterProtocolStatus::filtered_sample_bytes_change

The incremental change in the number of user samples preemptively filtered by a local DataWriter due to Content-Filtered Topics since the last time the status was read.

6.23.2.9 DDS_LongLong DDS_DataWriterProtocolStatus::sent_heartbeat_count

The number of Heartbeats sent between a local DataWriter and matching remote DataReader.

Because periodic and piggyback heartbeats are sent to remote readers and their locators differently in different situations, when a reader has more than one locator, this count may be larger than expected, to reflect the sending of Heartbeats to the multiple locators.

6.23.2.10 DDS_LongLong DDS_DataWriterProtocolStatus::sent_heartbeat_count_change

The incremental change in the number of Heartbeats sent between a local DataWriter and matching remote DataReader since the last time the status was read.

6.23.2.11 DDS_LongLong DDS_DataWriterProtocolStatus::sent_heartbeat_bytes

The number of bytes of Heartbeats sent between a local DataWriter and matching remote DataReader.

Because periodic and piggyback heartbeats are sent to remote readers and their locators differently in different situations, when a reader has more than one locator, this count may be larger than expected, to reflect the sending of Heartbeats to the multiple locators.

6.23.2.12 DDS_LongLong DDS_DataWriterProtocolStatus::sent_heartbeat_bytes_change

The incremental change in the number of bytes of Heartbeats sent between a local DataWriter and matching remote DataReader since the last time the status was read.

6.23.2.13 DDS_LongLong DDS_DataWriterProtocolStatus::pulled_sample_count

The number of user samples pulled from local DataWriter by matching DataReaders.

Pulled samples are samples sent for repairs, for late joiners, and all samples sent by the local DataWriter when **DDS_DataWriterProtocolQosPolicy::push_on_write** (p. 537) is **DDS_BOOLEAN_FALSE** (p. 299).

For large data, counts whole samples, not fragments.

6.23.2.14 DDS_LongLong DDS_DataWriterProtocolStatus::pulled_sample_count_change

The incremental change in the number of user samples pulled from local DataWriter by matching DataReaders since the last time the status was read.

Pulled samples are samples sent for repairs, for late joiners, and all samples sent by the local DataWriter when **DDS_DataWriterProtocolQosPolicy::push_on_write** (p. 537) is **DDS_BOOLEAN_FALSE** (p. 299).

For large data, counts whole samples, not fragments.

6.23.2.15 DDS_LongLong DDS_DataWriterProtocolStatus::pulled_sample_bytes

The number of bytes of user samples pulled from local DataWriter by matching DataReaders.

Pulled samples are samples sent for repairs, for late joiners, and all samples sent by the local DataWriter when **DDS_DataWriterProtocolQosPolicy::push_on_write** (p. 537) is **DDS_BOOLEAN_FALSE** (p. 299).

For large data, counts bytes of whole samples, not fragments.

6.23.2.16 DDS_LongLong DDS_DataWriterProtocolStatus::pulled_sample_bytes_change

The incremental change in the number of bytes of user samples pulled from local DataWriter by matching DataReaders since the last time the status was read.

Pulled samples are samples sent for repairs, for late joiners, and all samples sent by the local DataWriter when **DDS_**

DataWriterProtocolQosPolicy::push_on_write (p. 537) is **DDS_BOOLEAN_FALSE** (p. 299).

For large data, counts bytes of whole samples, not fragments.

6.23.2.17 DDS_LongLong DDS_DataWriterProtocolStatus::received_ack_count

The number of ACKs from a remote DataReader received by a local DataWriter.

6.23.2.18 DDS_LongLong DDS_DataWriterProtocolStatus::received_ack_count_change

The incremental change in the number of ACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

6.23.2.19 DDS_LongLong DDS_DataWriterProtocolStatus::received_ack_bytes

The number of bytes of ACKs from a remote DataReader received by a local DataWriter.

6.23.2.20 DDS_LongLong DDS_DataWriterProtocolStatus::received_ack_bytes_change

The incremental change in the number of bytes of ACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

6.23.2.21 DDS_LongLong DDS_DataWriterProtocolStatus::received_nack_count

The number of NACKs from a remote DataReader received by a local DataWriter.

6.23.2.22 DDS_LongLong DDS_DataWriterProtocolStatus::received_nack_count_change

The incremental change in the number of NACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

6.23.2.23 DDS_LongLong DDS_-DataWriterProtocolStatus::received_nack_bytes

The number of bytes of NACKs from a remote DataReader received by a local DataWriter.

6.23.2.24 DDS_LongLong DDS_-DataWriterProtocolStatus::received_nack_bytes_change

The incremental change in the number of bytes of NACKs from a remote DataReader received by a local DataWriter since the last time the status was read.

6.23.2.25 DDS_LongLong DDS_DataWriterProtocolStatus::sent_-gap_count

The number of GAPs sent from local DataWriter to matching remote DataReaders.

6.23.2.26 DDS_LongLong DDS_DataWriterProtocolStatus::sent_-gap_count_change

The incremental change in the number of GAPs sent from local DataWriter to matching remote DataReaders since the last time the status was read.

6.23.2.27 DDS_LongLong DDS_DataWriterProtocolStatus::sent_-gap_bytes

The number of bytes of GAPs sent from local DataWriter to matching remote DataReaders.

6.23.2.28 DDS_LongLong DDS_DataWriterProtocolStatus::sent_-gap_bytes_change

The incremental change in the number of bytes of GAPs sent from local DataWriter to matching remote DataReaders since the last time the status was read.

6.23.2.29 DDS_LongLong DDS_-DataWriterProtocolStatus::rejected_sample_count

The number of times a sample is rejected due to exceptions in the send path.

6.23.2.30 DDS_LongLong DDS_DataWriterProtocolStatus::rejected_sample_count_change

The incremental change in the number of times a sample is rejected due to exceptions in the send path since the last time the status was read.

6.23.2.31 DDS_Long DDS_DataWriterProtocolStatus::send_window_size

Current maximum number of outstanding samples allowed in the DataWriter's queue.

Spans the range from **DDS_RtpsReliableWriterProtocol_t::min_send_window_size** (p. 901) to **DDS_RtpsReliableWriterProtocol_t::max_send_window_size** (p. 901).

6.23.2.32 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::first_available_sample_sequence_number [read]

The sequence number of the first available sample currently queued in the local DataWriter.

Applies only for local DataWriter status.

6.23.2.33 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::last_available_sample_sequence_number [read]

The sequence number of the last available sample currently queued in the local DataWriter.

Applies only for local DataWriter status.

6.23.2.34 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::first_unacknowledged_sample_sequence_number [read]

The sequence number of the first unacknowledged sample currently queued in the local DataWriter.

Applies only for local DataWriter status.

6.23.2.35 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::first_available_sample_virtual_sequence_number [read]

The virtual sequence number of the first available sample currently queued in the local DataWriter.

Applies only for local DataWriter status.

6.23.2.36 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::last_available_sample_virtual_sequence_number [read]

The virtual sequence number of the last available sample currently queued in the local DataWriter.

Applies only for local DataWriter status.

6.23.2.37 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::first_unacknowledged_sample_virtual_sequence_number [read]

The virtual sequence number of the first unacknowledged sample currently queued in the local DataWriter.

Applies only for local DataWriter status.

6.23.2.38 DDS_InstanceHandle_t DDS_DataWriterProtocolStatus::first_unacknowledged_sample_subscription_handle

The handle of a remote DataReader that has not acknowledged the first unacknowledged sample of the local DataWriter.

Applies only for local DataWriter status.

6.23.2.39 struct DDS_SequenceNumber_t DDS_DataWriterProtocolStatus::first_unelapsed_keep_duration_sample_sequence_number [read]

The sequence number of the first sample whose keep duration has not yet elapsed.

Applicable only when **DDS_DataWriterProtocolQosPolicy::disable_positive_acks** (p. 537) is set.

Sequence number of the first sample kept in the DataWriter's queue whose keep_duration (applied when **DDS_DataWriterProtocolQosPolicy::disable_positive_acks** (p. 537) is set) has not yet elapsed.

Applies only for local DataWriter status.

6.24 DDS_DataWriterQos Struct Reference

QoS policies supported by a **DDSDataWriter** (p. 1113) entity.

Public Attributes

- struct **DDS_DurabilityQosPolicy durability**
  Durability policy, ***DURABILITY*** *(p. 348).*
- struct **DDS_DurabilityServiceQosPolicy durability_service**
  DurabilityService policy, ***DURABILITY_SERVICE*** *(p. 370).*
- struct **DDS_DeadlineQosPolicy deadline**
  Deadline policy, ***DEADLINE*** *(p. 353).*
- struct **DDS_LatencyBudgetQosPolicy latency_budget**
  Latency budget policy, ***LATENCY_BUDGET*** *(p. 354).*
- struct **DDS_LivelinessQosPolicy liveliness**
  Liveliness policy, ***LIVELINESS*** *(p. 358).*
- struct **DDS_ReliabilityQosPolicy reliability**
  Reliability policy, ***RELIABILITY*** *(p. 362).*
- struct **DDS_DestinationOrderQosPolicy destination_order**
  Destination order policy, ***DESTINATION_ORDER*** *(p. 365).*
- struct **DDS_HistoryQosPolicy history**
  History policy, ***HISTORY*** *(p. 367).*
- struct **DDS_ResourceLimitsQosPolicy resource_limits**
  Resource limits policy, ***RESOURCE_LIMITS*** *(p. 371).*
- struct **DDS_TransportPriorityQosPolicy transport_priority**
  Transport priority policy, ***TRANSPORT_PRIORITY*** *(p. 373).*
- struct **DDS_LifespanQosPolicy lifespan**
  Lifespan policy, ***LIFESPAN*** *(p. 374).*
- struct **DDS_UserDataQosPolicy user_data**
  User data policy, ***USER_DATA*** *(p. 345).*

- struct **DDS_OwnershipQosPolicy ownership**

  Ownership policy, ***OWNERSHIP*** *(p. 355).*

- struct **DDS_OwnershipStrengthQosPolicy ownership_strength**

  Ownership strength policy, ***OWNERSHIP_STRENGTH*** *(p. 357).*

- struct **DDS_WriterDataLifecycleQosPolicy writer_data_lifecycle**

  Writer data lifecycle policy, ***WRITER_DATA_LIFECYCLE*** *(p. 375).*

- struct **DDS_DataWriterResourceLimitsQosPolicy writer_resource_limits**

  <<**eXtension**>> (p. 199) *Writer resource limits policy,* ***DATA_WRITER_RESOURCE_LIMITS*** *(p. 409).*

- struct **DDS_DataWriterProtocolQosPolicy protocol**

  <<**eXtension**>> (p. 199) ***DDSDataWriter*** *(p. 1113) protocol policy,* ***DATA_WRITER_PROTOCOL*** *(p. 414)*

- struct **DDS_TransportSelectionQosPolicy transport_selection**

  <<**eXtension**>> (p. 199) *Transport plugin selection policy,* ***TRANSPORT_SELECTION*** *(p. 382).*

- struct **DDS_TransportUnicastQosPolicy unicast**

  <<**eXtension**>> (p. 199) *Unicast transport policy,* ***TRANSPORT_UNICAST*** *(p. 383).*

- struct **DDS_PublishModeQosPolicy publish_mode**

  <<**eXtension**>> (p. 199) *Publish mode policy,* ***PUBLISH_MODE*** *(p. 420).*

- struct **DDS_PropertyQosPolicy property**

  <<**eXtension**>> (p. 199) *Property policy,* ***PROPERTY*** *(p. 436).*

- struct **DDS_BatchQosPolicy batch**

  <<**eXtension**>> (p. 199) *Batch policy,* ***BATCH*** *(p. 431).*

- struct **DDS_MultiChannelQosPolicy multi_channel**

  <<**eXtension**>> (p. 199) *Multi channel policy,* ***MULTICHANNEL*** *(p. 435).*

- struct **DDS_AvailabilityQosPolicy availability**

  <<**eXtension**>> (p. 199) *Availability policy,* ***AVAILABILITY*** *(p. 442).*

- struct **DDS_EntityNameQosPolicy publication_name**

  <<**eXtension**>> (p. 199) *EntityName policy,* ***ENTITY_NAME*** *(p. 445).*

- struct **DDS_TypeSupportQosPolicy type_support**

  <<**eXtension**>> (p. 199) *Type support data,* ***TYPESUPPORT*** *(p. 429).*

6.24.1 Detailed Description

QoS policies supported by a **DDSDataWriter** (p. 1113) entity.

You must set certain members in a consistent manner:

- DDS_DataWriterQos::history.depth <= DDS_DataWriterQos::resource_limits.max_samples_per_instance

- DDS_DataWriterQos::resource_limits.max_samples_per_instance <= DDS_DataWriterQos::resource_limits.max_samples

- DDS_DataWriterQos::resource_limits.initial_samples <= DDS_DataWriterQos::resource_limits.max_samples

- DDS_DataWriterQos::resource_limits.initial_instances <= DDS_DataWriterQos::resource_limits.max_instances

- length of DDS_DataWriterQos::user_data.value <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .writer_user_data_max_length

If any of the above are not true, **DDSDataWriter::set_qos** (p. 1126) and **DDSDataWriter::set_qos_with_profile** (p. 1127) and **DDSPublisher::set_default_datawriter_qos** (p. 1351) and **DDSPublisher::set_default_datawriter_qos_with_profile** (p. 1351) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) and **DDSPublisher::create_datawriter** (p. 1355) and **DDSPublisher::create_datawriter_with_profile** (p. 1357) and will return NULL.

Entity:

DDSDataWriter (p. 1113)

See also:

QoS Policies (p. 331) allowed ranges within each Qos.

6.24.2 Member Data Documentation

6.24.2.1 struct DDS_DurabilityQosPolicy DDS_DataWriterQos::durability [read]

Durability policy, **DURABILITY** (p. 348).

6.24.2.2 struct DDS_DurabilityServiceQosPolicy DDS_DataWriterQos::durability_service [read]

DurabilityService policy, **DURABILITY_SERVICE** (p. 370).

6.24.2.3 struct DDS_DeadlineQosPolicy DDS_DataWriterQos::deadline [read]

Deadline policy, **DEADLINE** (p. 353).

6.24.2.4 struct DDS_LatencyBudgetQosPolicy DDS_DataWriterQos::latency_budget [read]

Latency budget policy, **LATENCY_BUDGET** (p. 354).

6.24.2.5 struct DDS_LivelinessQosPolicy DDS_DataWriterQos::liveliness [read]

Liveliness policy, **LIVELINESS** (p. 358).

6.24.2.6 struct DDS_ReliabilityQosPolicy DDS_DataWriterQos::reliability [read]

Reliability policy, **RELIABILITY** (p. 362).

6.24.2.7 struct DDS_DestinationOrderQosPolicy DDS_DataWriterQos::destination_order [read]

Destination order policy, **DESTINATION_ORDER** (p. 365).

6.24.2.8 struct DDS_HistoryQosPolicy DDS_DataWriterQos::history [read]

History policy, **HISTORY** (p. 367).

6.24.2.9 struct DDS_ResourceLimitsQosPolicy DDS_DataWriterQos::resource_limits [read]

Resource limits policy, **RESOURCE_LIMITS** (p. 371).

6.24.2.10 struct DDS_TransportPriorityQosPolicy DDS_DataWriterQos::transport_priority [read]

Transport priority policy, **TRANSPORT_PRIORITY** (p. 373).

6.24.2.11 struct DDS_LifespanQosPolicy DDS_DataWriterQos::lifespan [read]

Lifespan policy, **LIFESPAN** (p. 374).

6.24.2.12 struct DDS_UserDataQosPolicy DDS_DataWriterQos::user_data [read]

User data policy, **USER_DATA** (p. 345).

6.24.2.13 struct DDS_OwnershipQosPolicy DDS_DataWriterQos::ownership [read]

Ownership policy, **OWNERSHIP** (p. 355).

6.24.2.14 struct DDS_OwnershipStrengthQosPolicy DDS_DataWriterQos::ownership_strength [read]

Ownership strength policy, **OWNERSHIP_STRENGTH** (p. 357).

6.24.2.15 struct DDS_WriterDataLifecycleQosPolicy DDS_DataWriterQos::writer_data_lifecycle [read]

Writer data lifecycle policy, **WRITER_DATA_LIFECYCLE** (p. 375).

6.24.2.16 struct DDS_DataWriterResourceLimitsQosPolicy DDS_DataWriterQos::writer_resource_limits [read]

<<*eXtension*>> *(p. 199)* Writer resource limits policy, **DATA_WRITER_RESOURCE_LIMITS** (p. 409).

6.24.2.17 struct DDS_DataWriterProtocolQosPolicy DDS_DataWriterQos::protocol [read]

<<*eXtension*>> *(p. 199)* **DDSDataWriter** (p. 1113) protocol policy, **DATA_WRITER_PROTOCOL** (p. 414)

6.24.2.18 struct DDS_TransportSelectionQosPolicy DDS_DataWriterQos::transport_selection [read]

<<*eXtension*>> *(p. 199)* Transport plugin selection policy, **TRANSPORT_SELECTION** (p. 382).

Specifies the transports available for use by the **DDSDataWriter** (p. 1113).

6.24.2.19 struct DDS_TransportUnicastQosPolicy DDS_DataWriterQos::unicast [read]

<<*eXtension*>> *(p. 199)* Unicast transport policy, **TRANSPORT_UNICAST** (p. 383).

Specifies the unicast transport interfaces and ports on which **messages** can be received.

The unicast interfaces are used to receive messages from **DDSDataReader** (p. 1087) entities in the domain.

6.24.2.20 struct DDS_PublishModeQosPolicy DDS_DataWriterQos::publish_mode [read]

<<*eXtension*>> *(p. 199)* Publish mode policy, **PUBLISH_MODE** (p. 420).

Determines whether the **DDSDataWriter** (p. 1113) publishes data synchronously or asynchronously and how.

6.24.2.21 struct DDS_PropertyQosPolicy DDS_DataWriterQos::property [read]

<<*eXtension*>> *(p. 199)* Property policy, **PROPERTY** (p. 436).

6.24.2.22 struct DDS_BatchQosPolicy DDS_DataWriterQos::batch [read]

<<*eXtension*>> *(p. 199)* Batch policy, **BATCH** (p. 431).

6.24.2.23 struct DDS_MultiChannelQosPolicy DDS_DataWriterQos::multi_channel [read]

<<*eXtension*>> *(p. 199)* Multi channel policy, **MULTICHANNEL** (p. 435).

6.24.2.24 struct DDS_AvailabilityQosPolicy DDS_DataWriterQos::availability [read]

<<*eXtension*>> *(p. 199)* Availability policy, **AVAILABILITY** (p. 442).

6.24.2.25 struct DDS_EntityNameQosPolicy DDS_DataWriterQos::publication_name [read]

<<*eXtension*>> *(p. 199)* EntityName policy, **ENTITY_NAME** (p. 445).

6.24.2.26 struct DDS_TypeSupportQosPolicy DDS_DataWriterQos::type_support [read]

<<*eXtension*>> *(p. 199)* Type support data, **TYPESUPPORT** (p. 429).

Optional value that is passed to a type plugin's on_endpoint_attached and serialization functions.

6.25 DDS_DataWriterResourceLimitsQosPolicy Struct Reference

Various settings that configure how a **DDSDataWriter** (p. 1113) allocates and uses physical memory for internal resources.

Public Attributes

- **DDS_Long initial_concurrent_blocking_threads**

  *The initial number of threads that are allowed to concurrently block on write call on the same **DDSDataWriter** (p. 1113).*

- **DDS_Long max_concurrent_blocking_threads**

  *The maximum number of threads that are allowed to concurrently block on write call on the same **DDSDataWriter** (p. 1113).*

- **DDS_Long max_remote_reader_filters**

  *The maximum number of remote DataReaders for which the **DDSDataWriter** (p. 1113) will perform content-based filtering.*

- **DDS_Long initial_batches**

  *Represents the initial number of batches a **DDSDataWriter** (p. 1113) will manage.*

- **DDS_Long max_batches**

  *Represents the maximum number of batches a **DDSDataWriter** (p. 1113) will manage.*

- **DDS_DataWriterResourceLimitsInstanceReplacementKind instance_replacement**

  Sets the kinds of instances allowed to be replaced when instance resource limits are reached.

- **DDS_Boolean replace_empty_instances**

  Whether or not to replace empty instances during instance replacement.

- **DDS_Boolean autoregister_instances**

  Whether or not to automatically register new instances.

- **DDS_Long initial_virtual_writers**

  *The initial number of virtual writers supported by a **DDSDataWriter** (p. 1113).*

- **DDS_Long max_virtual_writers**

  *The maximum number of virtual writers supported by a **DDSDataWriter** (p. 1113).*

- **DDS_Long max_remote_readers**

  *The maximum number of remote readers supported by a **DDSDataWriter** (p. 1113).*

- **DDS_Long max_app_ack_remote_readers**

  *The maximum number of application-level acknowledging remote readers supported by a **DDSDataWriter** (p. 1113).*

6.25.1 Detailed Description

Various settings that configure how a **DDSDataWriter** (p. 1113) allocates and uses physical memory for internal resources.

DataWriters must allocate internal structures to handle the simultaneously blocking of threads trying to call **FooDataWriter::write** (p. 1484) on the same **DDSDataWriter** (p. 1113), for the storage used to batch small samples, and for content-based filters specified by DataReaders.

Most of these internal structures start at an initial size and, by default, will be grown as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want to bound the amount of memory that can be used by a **DDSDataWriter** (p. 1113). By setting the initial size to the maximum size, you will prevent RTI Connext from dynamically allocating any memory after the creation of the **DDSDataWriter** (p. 1113).

This QoS policy is an extension to the DDS standard.

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.25.2 Member Data Documentation

6.25.2.1 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::initial_concurrent_blocking_threads

The initial number of threads that are allowed to concurrently block on write call on the same **DDSDataWriter** (p. 1113).

This value only applies if **DDS_HistoryQosPolicy** (p. 758) has its kind set to **DDS_KEEP_ALL_HISTORY_QOS** (p. 368) and **DDS_ReliabilityQosPolicy::max_blocking_time** (p. 868) is > 0.

[default] 1

[range] [1, 10000], <= max_concurrent_blocking_threads

6.25.2.2 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::max_concurrent_blocking_threads

The maximum number of threads that are allowed to concurrently block on write call on the same **DDSDataWriter** (p. 1113).

This value only applies if **DDS_HistoryQosPolicy** (p. 758) has its kind set to **DDS_KEEP_ALL_HISTORY_QOS** (p. 368) and **DDS_ReliabilityQosPolicy::max_blocking_time** (p. 868) is > 0.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 10000] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_concurrent_blocking_threads

6.25.2.3 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::max_remote_reader_filters

The maximum number of remote DataReaders for which the **DDSDataWriter** (p. 1113) will perform content-based filtering.

[default] 32

[range] [0, (2^31)-2] or **DDS_LENGTH_UNLIMITED** (p. 371).

0: The **DDSDataWriter** (p. 1113) will not perform filtering for any **DDSDataReader** (p. 1087).

1 to (2^31)-2: The DataWriter will filter for up to the specified number of DataReaders. In addition, the Datawriter will store the result of the filtering per sample per DataReader.

DDS_LENGTH_UNLIMITED (p. 371): The DataWriter will filter for up to (2^31)-2 DataReaders. However, in this case, the DataWriter will not store the filtering result per sample per DataReader. Thus, if a sample is resent (such as due to a loss of reliable communication), the sample will be filtered again.

6.25.2.4 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::initial_batches

Represents the initial number of batches a **DDSDataWriter** (p. 1113) will manage.

[**default**] 8

[**range**] [1,100 million]

See also:

DDS_BatchQosPolicy (p. 476)

6.25.2.5 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::max_batches

Represents the maximum number of batches a **DDSDataWriter** (p. 1113) will manage.

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

When batching is enabled, the maximum number of samples that a **DDSDataWriter** (p. 1113) can store is limited by this value and **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881).

[**range**] [1,100 million] or **DDS_LENGTH_UNLIMITED** (p. 371) >= **DDS_RtpsReliableWriterProtocol_t::heartbeats_per_max_samples** (p. 896) if batching is enabled

See also:

DDS_BatchQosPolicy (p. 476)

6.25.2.6 DDS_DataWriterResourceLimitsInstanceReplacementKind DDS_DataWriterResourceLimitsQosPolicy::instance_replacement

Sets the kinds of instances allowed to be replaced when instance resource limits are reached.

When a **DDSDataWriter** (p. 1113)'s number of active instances is greater than **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882), it will try to make room by replacing an existing instance. This field specifies the kinds of instances allowed to be replaced.

If a replaceable instance is not available, either an out-of-resources exception will be returned, or the writer may block if the instance reclamation was done when writing.

[default] **DDS_UNREGISTERED_INSTANCE_REPLACEMENT** (p. 411)

See also:

DDS_DataWriterResourceLimitsInstanceReplacementKind (p. 410)

6.25.2.7 DDS_Boolean DDS_DataWriterResourceLimitsQosPolicy::replace_empty_instances

Whether or not to replace empty instances during instance replacement.

When a **DDSDataWriter** (p. 1113) has more active instances than allowed by **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882), it tries to make room by replacing an existing instance. This field configures whether empty instances (i.e. instances with no samples) may be replaced. If set **DDS_BOOLEAN_TRUE** (p. 298), then a **DDSDataWriter** (p. 1113) will first try reclaiming empty instances, before trying to replace whatever is specified by **DDS_DataWriterResourceLimitsQosPolicy::instance_replacement** (p. 563).

[default] **DDS_BOOLEAN_FALSE** (p. 299)

See also:

DDS_DataWriterResourceLimitsInstanceReplacementKind (p. 410)

6.25.2.8 DDS_Boolean DDS_DataWriterResourceLimitsQosPolicy::autoregister_instances

Whether or not to automatically register new instances.

[default] **DDS_BOOLEAN_TRUE** (p. 298)

When set to true, it is possible to write with a non-NIL handle of an instance that is not registered: the write operation will succeed and the instance will be registered. Otherwise, that write operation would fail.

See also:

FooDataWriter::write (p. 1484)

6.25.2.9 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::initial_virtual_writers

The initial number of virtual writers supported by a **DDSDataWriter** (p. 1113).

[**default**] 1

[**range**] [1, 1000000], or **DDS_LENGTH_UNLIMITED** (p. 371)

6.25.2.10 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::max_virtual_writers

The maximum number of virtual writers supported by a **DDSDataWriter** (p. 1113).

Sets the maximum number of unique virtual writers supported by a **DDSDataWriter** (p. 1113), where virtual writers are added when samples are written with the virtual writer GUID.

This field is specially relevant in the configuration of Persistence Service DataWriters since these DataWriters will publish samples on behalf of multiple virtual writers.

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

[**range**] [1, 1000000], or **DDS_LENGTH_UNLIMITED** (p. 371)

6.25.2.11 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::max_remote_readers

The maximum number of remote readers supported by a **DDSDataWriter** (p. 1113).

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

[**range**] [1, 1000000], or **DDS_LENGTH_UNLIMITED** (p. 371)

6.25.2.12 DDS_Long DDS_DataWriterResourceLimitsQosPolicy::max_app_ack_remote_readers

The maximum number of application-level acknowledging remote readers supported by a **DDSDataWriter** (p. 1113).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1000000], or **DDS_LENGTH_UNLIMITED** (p. 371)

6.26 DDS_DeadlineQosPolicy Struct Reference

Expresses the maximum duration (deadline) within which an instance is expected to be updated.

Public Attributes

- struct **DDS_Duration_t period**
  Duration of the deadline period.

6.26.1 Detailed Description

Expresses the maximum duration (deadline) within which an instance is expected to be updated.

A **DDSDataReader** (p. 1087) expects a new sample updating the value of each instance at least once every `period`. That is, `period` specifies the maximum expected elapsed time between arriving data samples.

A **DDSDataWriter** (p. 1113) indicates that the application commits to write a new value (using the **DDSDataWriter** (p. 1113)) for each instance managed by the **DDSDataWriter** (p. 1113) at least once every `period`.

This QoS can be used during system integration to ensure that applications have been coded to meet design specifications.

It can also be used during run time to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions to prevent total system failure when data is not received in time. For topics on which data is not expected to be periodic, `period` should be set to an infinite value.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_OFFERED_DEADLINE_MISSED_STATUS (p. 323), **DDS_REQUESTED_DEADLINE_MISSED_STATUS** (p. 323), **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES

Changeable (p. 340) = **YES** (p. 340)

6.26.2 Usage

This policy is useful for cases where a **DDSTopic** (p. 1419) is expected to have each instance updated periodically. On the publishing side this setting establishes a contract that the application must meet. On the subscribing side the setting establishes a minimum requirement for the remote publishers that are expected to supply the data values.

When RTI Connext 'matches' a **DDSDataWriter** (p. 1113) and a **DDSDataReader** (p. 1087) it checks whether the settings are compatible (i.e., *offered deadline <= requested deadline*); if they are not, the two entities are informed (via the **DDSListener** (p. 1318) or **DDSCondition** (p. 1075) mechanism) of the incompatibility of the QoS settings and communication will not occur.

Assuming that the reader and writer ends have compatible settings, the fulfilment of this contract is monitored by RTI Connext and the application is informed of any violations by means of the proper **DDSListener** (p. 1318) or **DDSCondition** (p. 1075).

6.26.3 Compatibility

The value offered is considered compatible with the value requested if and only if the inequality *offered period <= requested period* holds.

6.26.4 Consistency

The setting of the **DEADLINE** (p. 353) policy must be set consistently with that of the **TIME_BASED_FILTER** (p. 360).

For these two policies to be consistent the settings must be such that *deadline period >= minimum_separation*.

An attempt to set these policies in an inconsistent manner will result in **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) in **set_qos (abstract)** (p. 1254), or the **DDSEntity** (p. 1253) will not be created.

For a **DDSDataReader** (p. 1087), the **DEADLINE** (p. 353) policy and **DDS_TimeBasedFilterQosPolicy** (p. 954) may interact such that even though the **DDSDataWriter** (p. 1113) is writing samples fast enough to fulfill its commitment to its own deadline, the **DDSDataReader** (p. 1087) may see violations of its deadline. This happens because RTI Connext will drop any samples received within the **DDS_TimeBasedFilterQosPolicy::minimum_separation** (p. 957). To avoid triggering the **DDSDataReader** (p. 1087)'s

deadline, even though the matched **DDSDataWriter** (p. 1113) is meeting its own deadline, set the two QoS parameters so that:

reader deadline >= reader minimum_separation + writer deadline

See **DDS_TimeBasedFilterQosPolicy** (p. 954) for more information about the interactions between deadlines and time-based filters.

See also:

DDS_TimeBasedFilterQosPolicy (p. 954)

6.26.5 Member Data Documentation

6.26.5.1 struct DDS_Duration_t DDS_DeadlineQosPolicy::period [read]

Duration of the deadline period.

[**default**] **DDS_DURATION_INFINITE** (p. 305)

[**range**] [1 nanosec, 1 year] or **DDS_DURATION_INFINITE** (p. 305), >= **DDS_TimeBasedFilterQosPolicy::minimum_separation** (p. 957)

6.27 DDS_DestinationOrderQosPolicy Struct Reference

Controls how the middleware will deal with data sent by multiple **DDSDataWriter** (p. 1113) entities for the same instance of data (i.e., same **DDSTopic** (p. 1419) and key).

Public Attributes

- **DDS_DestinationOrderQosPolicyKind kind**

  Specifies the desired kind of destination order.

- struct **DDS_Duration_t source_timestamp_tolerance**

  <<**eXtension**>> (p. 199) *Allowed tolerance between source timestamps of consecutive samples.*

6.27.1 Detailed Description

Controls how the middleware will deal with data sent by multiple **DDSDataWriter** (p. 1113) entities for the same instance of data (i.e., same **DDSTopic** (p. 1419) and key).

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.27.2 Usage

When multiple DataWriters send data for the same topic, the order in which data from different DataWriters are received by the applications of different

DataReaders may be different. So different DataReaders may not receive the same "last" value when DataWriters stop sending data.

This QoS policy controls how each subscriber resolves the final value of a data instance that is written by multiple **DDSDataWriter** (p. 1113) entities (which may be associated with different **DDSPublisher** (p. 1346) entities) running on different nodes.

The default setting, **DDS_BY_RECEPTION_TIMESTAMP_-DESTINATIONORDER_QOS** (p. 366), indicates that (assuming the **OWNERSHIP_STRENGTH** (p. 357) policy allows it) the latest received value for the instance should be the one whose value is kept. That is, data will be delivered by a **DDSDataReader** (p. 1087) in the order in which it was *received* (which may lead to inconsistent final values).

The setting **DDS_BY_SOURCE_TIMESTAMP_-DESTINATIONORDER_QOS** (p. 366) indicates that (assuming the **OWNERSHIP_STRENGTH** (p. 357) allows it, within each instance) the `source_timestamp` of the change shall be used to determine the most recent information. That is, data will be delivered by a **DDSDataReader** (p. 1087) in the order in which it was *sent*. If data arrives on the network with a source timestamp that is later than the source timestamp of the last data delivered, the new data will be dropped. This 'by source timestamp' ordering therefore works best when system clocks are relatively synchronized among writing machines.

When using **DDS_BY_SOURCE_TIMESTAMP_-DESTINATIONORDER_QOS** (p. 366), not all data sent by multiple **DDSDataWriter** (p. 1113) entities may be delivered to a **DDSDataReader** (p. 1087) and not all DataReaders will see the same data sent by DataWriters. However, all DataReaders will see the same "final" data when DataWriters "stop" sending data. This is the only setting that, in the case of concurrently publishing **DDSDataWriter** (p. 1113) entities updating the same instance of a shared-ownership topic, ensures all subscribers will end up with the same final value for the instance.

This QoS can be used to create systems that have the property of "eventual consistency." Thus intermediate states across multiple applications may be inconsistent, but when DataWriters stop sending changes to the same topic, all applications will end up having the same state.

6.27.3 Compatibility

The value offered is considered compatible with the value requested if and only if the inequality *offered kind* >= *requested kind* evaluates to 'TRUE'. For the purposes of this inequality, the values of **DDS_DestinationOrderQosPolicy::kind** (p. 572) are consid-

ered ordered such that **DDS_BY_RECEPTION_TIMESTAMP_-DESTINATIONORDER_QOS** (p. 366) < **DDS_BY_SOURCE_-TIMESTAMP_DESTINATIONORDER_QOS** (p. 366)

6.27.4 Member Data Documentation

6.27.4.1 DDS_DestinationOrderQosPolicyKind DDS_DestinationOrderQosPolicy::kind

Specifies the desired kind of destination order.

[default] **DDS_BY_RECEPTION_TIMESTAMP_-DESTINATIONORDER_QOS** (p. 366),

6.27.4.2 struct DDS_Duration_t DDS_-DestinationOrderQosPolicy::source_timestamp_tolerance [read]

<<***eXtension***>> *(p. 199)* Allowed tolerance between source timestamps of consecutive samples.

When a **DDSDataWriter** (p. 1113) sets **DDS_-DestinationOrderQosPolicyKind** (p. 366) to **DDS_BY_SOURCE_-TIMESTAMP_DESTINATIONORDER_QOS** (p. 366), when when writing a sample, its timestamp must not be less than the timestamp of the previously written sample. However, if it is less than the timestamp of the previously written sample but the difference is less than this tolerance, the sample will use the previously written sample's timestamp as its timestamp. Otherwise, if the difference is greater than this tolerance, the write will fail.

When a **DDSDataReader** (p. 1087) sets **DDS_-DestinationOrderQosPolicyKind** (p. 366) to **DDS_BY_SOURCE_-TIMESTAMP_DESTINATIONORDER_QOS** (p. 366), the **DDS-DataReader** (p. 1087) will accept a sample only if the difference between its source timestamp and the reception timestamp is no greater than this tolerance. Otherwise, the sample is rejected.

[default] 100 milliseconds for **DDSDataWriter** (p. 1113), 30 seconds for **DDSDataReader** (p. 1087)

6.28 DDS_DiscoveryConfigQosPolicy Struct Reference

Settings for discovery configuration.

Public Attributes

- struct **DDS_Duration_t participant_liveliness_lease_duration**

  The liveliness lease duration for the participant.

- struct **DDS_Duration_t participant_liveliness_assert_period**

  The period to assert liveliness for the participant.

- **DDS_RemoteParticipantPurgeKind remote_participant_purge_kind**

  The participant's behavior for maintaining knowledge of remote participants (and their contained entities) with which discovery communication has been lost.

- struct **DDS_Duration_t max_liveliness_loss_detection_period**

  The maximum amount of time between when a remote entity stops maintaining its liveliness and when the matched local entity realizes that fact.

- **DDS_Long initial_participant_announcements**

  The number of initial announcements sent when a participant is first enabled or when a remote participant is newly discovered.

- struct **DDS_Duration_t min_initial_participant_announcement_period**

  The minimum period between initial announcements when a participant is first enabled or when a remote participant is newly discovered.

- struct **DDS_Duration_t max_initial_participant_announcement_period**

  The maximum period between initial announcements when a participant is first enabled or when a remote participant is newly discovered.

- struct **DDS_BuiltinTopicReaderResourceLimits_t participant_reader_resource_limits**

  Resource limits.

- struct **DDS_RtpsReliableReaderProtocol_t publication_reader**

  RTPS protocol-related configuration settings for a built-in publication reader.

- struct **DDS_BuiltinTopicReaderResourceLimits_t publication_reader_resource_limits**

  Resource limits.

- struct **DDS_RtpsReliableReaderProtocol_t subscription_reader**

  RTPS protocol-related configuration settings for a built-in subscription reader.

- struct **DDS_BuiltinTopicReaderResourceLimits_t subscription_reader_resource_limits**

  Resource limits.

- struct **DDS_RtpsReliableWriterProtocol_t publication_writer**

  RTPS protocol-related configuration settings for a built-in publication writer.

- struct **DDS_WriterDataLifecycleQosPolicy publication_writer_data_lifecycle**

  Writer data lifecycle settings for a built-in publication writer.

- struct **DDS_RtpsReliableWriterProtocol_t subscription_writer**

  RTPS protocol-related configuration settings for a built-in subscription writer.

- struct **DDS_WriterDataLifecycleQosPolicy subscription_writer_data_lifecycle**

  Writer data lifecycle settings for a built-in subscription writer.

- **DDS_DiscoveryConfigBuiltinPluginKindMask builtin_discovery_plugins**

  The kind mask for built-in discovery plugins.

- struct **DDS_RtpsReliableReaderProtocol_t participant_message_reader**

  RTPS protocol-related configuration settings for a built-in participant message reader.

- struct **DDS_RtpsReliableWriterProtocol_t participant_message_writer**

  RTPS protocol-related configuration settings for a built-in participant message writer.

- struct **DDS_PublishModeQosPolicy publication_writer_publish_mode**

<<**eXtension**>> (p. 199) *Publish mode policy,* ***PUBLISH_MODE*** *(p. 420).*

- struct **DDS_AsynchronousPublisherQosPolicy asynchronous_publisher**

  <<**eXtension**>> (p. 199) *Asynchronous publishing settings for the Discovery* ***DDSPublisher*** *(p. 1346) and all entities that are created by it.*

6.28.1 Detailed Description

Settings for discovery configuration.

This QoS policy is an extension to the DDS standard.

This QoS policy controls the amount of delay in discovering entities in the system and the amount of discovery traffic in the network.

The amount of network traffic required by the discovery process can vary widely, based on how your application has chosen to configure the middleware's network addressing (e.g., unicast vs. multicast, multicast TTL, etc.), the size of the system, whether all applications are started at the same time or whether start times are staggered, and other factors. Your application can use this policy to make tradeoffs between discovery completion time and network bandwidth utilization. In addition, you can introduce random back-off periods into the discovery process to decrease the probability of network contention when many applications start simultaneously.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.28.2 Member Data Documentation

6.28.2.1 struct DDS_Duration_t DDS_DiscoveryConfigQosPolicy::participant_liveliness_lease_duration [read]

The liveliness lease duration for the participant.

This is the same as the expiration time of the DomainParticipant as defined in the RTPS protocol.

If the participant has not refreshed its own liveliness to other participants at least once within this period, it may be considered as stale by other participants in the network.

Should be strictly greater than **DDS_DiscoveryConfigQosPolicy::participant_liveliness_assert_period** (p. 576).

[default] 100 seconds

[range] [1 nanosec,1 year], > participant_liveliness_assert_period

6.28.2.2 struct DDS_Duration_t DDS_DiscoveryConfigQosPolicy::participant_liveliness_assert_period [read]

The period to assert liveliness for the participant.

The period at which the participant will refresh its liveliness to all the peers.

Should be strictly less than **DDS_DiscoveryConfigQosPolicy::participant_liveliness_lease_duration** (p. 575).

[default] 30 seconds

[range] [1 nanosec,1 year), < participant_liveliness_lease_duration

6.28.2.3 DDS_RemoteParticipantPurgeKind DDS_DiscoveryConfigQosPolicy::remote_participant_purge_kind

The participant's behavior for maintaining knowledge of remote participants (and their contained entities) with which discovery communication has been lost.

Most users will not need to change this value from its default, **DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE** (p. 426). However, **DDS_NO_REMOTE_PARTICIPANT_PURGE** (p. 426) may be a good choice if the following conditions apply:

1. Discovery communication with a remote participant may be lost while data communication remains intact. Such will not typically be the case if discovery takes place over the Simple Discovery Protocol, but may be the case if the RTI Enterprise Discovery Service is used.

2. Extensive and prolonged lack of discovery communication between participants is not expected to be common, either because participant loss itself is expected to be rare, or because participants may be lost sporadically but will typically return again.

3. Maintaining inter-participant liveliness is problematic, perhaps because a participant has no writers with the appropriate **DDS_LivelinessQosPolicyKind** (p. 358).

[default] **DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE** (p. 426)

6.28.2.4 struct DDS_Duration_t DDS_DiscoveryConfigQosPolicy::max_liveliness_loss_detection_period [read]

The maximum amount of time between when a remote entity stops maintaining its liveliness and when the matched local entity realizes that fact.

Notification of the loss of liveliness of a remote entity may come more quickly that this duration, depending on the liveliness contract between the local and remote entities and the capabilities of the discovery mechanism in use. For example, a **DDSDataReader** (p. 1087) will learn of the loss of liveliness of a matched **DDSDataWriter** (p. 1113) within the reader's offered liveliness lease duration.

Shortening this duration will increase the responsiveness of entities to communication failures. However, it will also increase the CPU usage of the application, as the liveliness of remote entities will be examined more frequently.

[default] 60 seconds

[range] [0, 1 year]

6.28.2.5 DDS_Long DDS_DiscoveryConfigQosPolicy::initial_participant_announcements

The number of initial announcements sent when a participant is first enabled or when a remote participant is newly discovered.

Also, when a new remote participant appears, the local participant can announce itself to the peers multiple times controlled by this parameter.

[default] 5

[range] [0,1 million]

6.28.2.6 struct DDS_Duration_t DDS_DiscoveryConfigQosPolicy::min_initial_participant_announcement_period [read]

The minimum period between initial announcements when a participant is first enabled or when a remote participant is newly discovered.

A random delay between this and **DDS_DiscoveryConfigQosPolicy::max_initial_participant_announcement_period** (p. 578) is introduced in between initial announcements when a new remote participant is discovered.

The setting of **DDS_DiscoveryConfigQosPolicy::min_initial_participant_announcement_period** (p. 577) must be consistent with **DDS_DiscoveryConfigQosPolicy::max_initial_participant_announcement_period** (p. 578). For these two values to be consistent, they must verify that:

DDS_DiscoveryConfigQosPolicy::min_initial_participant_announcement_period (p. 577) <= **DDS_DiscoveryConfigQosPolicy::max_initial_participant_announcement_period** (p. 578).

[default] 1 second

[range] [1 nanosec,1 year]

6.28.2.7 struct DDS_Duration_t DDS_DiscoveryConfigQosPolicy::max_initial_participant_announcement_period [read]

The maximum period between initial announcements when a participant is first enabled or when a remote participant is newly discovered.

A random delay between **DDS_DiscoveryConfigQosPolicy::min_initial_participant_announcement_period** (p. 577) and this is introduced in between initial announcements when a new remote participant is discovered.

The setting of **DDS_DiscoveryConfigQosPolicy::max_initial_participant_announcement_period** (p. 578) must be consistent with **DDS_DiscoveryConfigQosPolicy::min_initial_participant_announcement_period** (p. 577). For these two values to be consistent, they must verify that:

DDS_DiscoveryConfigQosPolicy::min_initial_participant_announcement_period (p. 577) <= **DDS_DiscoveryConfigQosPolicy::max_initial_participant_announcement_period** (p. 578).

[default] 1 second

[range] [1 nanosec,1 year]

6.28.2.8 struct DDS_BuiltinTopicReaderResourceLimits_t DDS_DiscoveryConfigQosPolicy::participant_reader_resource_limits [read]

Resource limits.

Resource limit of the built-in topic participant reader. For details, see **DDS_BuiltinTopicReaderResourceLimits_t** (p. 482).

6.28.2.9 struct DDS_RtpsReliableReaderProtocol_t DDS_DiscoveryConfigQosPolicy::publication_reader [read]

RTPS protocol-related configuration settings for a built-in publication reader.

For details, refer to the **DDS_DataReaderQos** (p. 515)

6.28.2.10 struct DDS_BuiltinTopicReaderResourceLimits_t DDS_DiscoveryConfigQosPolicy::publication_reader_resource_limits [read]

Resource limits.

Resource limit of the built-in topic publication reader. For details, see **DDS_BuiltinTopicReaderResourceLimits_t** (p. 482).

6.28.2.11 struct DDS_RtpsReliableReaderProtocol_t DDS_DiscoveryConfigQosPolicy::subscription_reader [read]

RTPS protocol-related configuration settings for a built-in subscription reader.

For details, refer to the **DDS_DataReaderQos** (p. 515)

6.28.2.12 struct DDS_BuiltinTopicReaderResourceLimits_t DDS_DiscoveryConfigQosPolicy::subscription_reader_resource_limits [read]

Resource limits.

Resource limit of the built-in topic subscription reader. For details, see **DDS_BuiltinTopicReaderResourceLimits_t** (p. 482).

6.28.2.13 struct DDS_RtpsReliableWriterProtocol_t DDS_DiscoveryConfigQosPolicy::publication_writer [read]

RTPS protocol-related configuration settings for a built-in publication writer.

For details, refer to the **DDS_DataWriterQos** (p. 553)

6.28.2.14 struct DDS_WriterDataLifecycleQosPolicy DDS_DiscoveryConfigQosPolicy::publication_writer_data_lifecycle [read]

Writer data lifecycle settings for a built-in publication writer.

For details, refer to the **DDS_WriterDataLifecycleQosPolicy** (p. 1071). **DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances** (p. 1072) will always be forced to **DDS_BOOLEAN_TRUE** (p. 298).

6.28.2.15 struct DDS_RtpsReliableWriterProtocol_t DDS_DiscoveryConfigQosPolicy::subscription_writer [read]

RTPS protocol-related configuration settings for a built-in subscription writer.

For details, refer to the **DDS_DataWriterQos** (p. 553)

6.28.2.16 struct DDS_WriterDataLifecycleQosPolicy DDS_DiscoveryConfigQosPolicy::subscription_writer_data_lifecycle [read]

Writer data lifecycle settings for a built-in subscription writer.

For details, refer to the **DDS_WriterDataLifecycleQosPolicy** (p. 1071). **DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances** (p. 1072) will always be forced to **DDS_BOOLEAN_TRUE** (p. 298).

6.28.2.17 DDS_DiscoveryConfigBuiltinPluginKindMask DDS_DiscoveryConfigQosPolicy::builtin_discovery_plugins

The kind mask for built-in discovery plugins.

There are several built-in discovery plugin. This mask enables the different plugins. Any plugin not enabled will not be created.

[default] **DDS_DISCOVERYCONFIG_BUILTIN_SDP** (p. 425)

6.28.2.18 struct DDS_RtpsReliableReaderProtocol_t DDS_DiscoveryConfigQosPolicy::participant_message_reader [read]

RTPS protocol-related configuration settings for a built-in participant message reader.

For details, refer to the **DDS_DataReaderQos** (p. 515)

6.28.2.19 struct DDS_RtpsReliableWriterProtocol_t DDS_DiscoveryConfigQosPolicy::participant_message_writer [read]

RTPS protocol-related configuration settings for a built-in participant message writer.

For details, refer to the **DDS_DataWriterQos** (p. 553)

6.28.2.20 struct DDS_PublishModeQosPolicy DDS_DiscoveryConfigQosPolicy::publication_writer_publish_mode [read]

<<***eXtension***>> *(p. 199)* Publish mode policy, **PUBLISH_MODE** (p. 420).

Determines whether the Discovery **DDSDataWriter** (p. 1113) publishes data synchronously or asynchronously and how.

6.28.2.21 struct DDS_AsynchronousPublisherQosPolicy DDS_DiscoveryConfigQosPolicy::asynchronous_publisher [read]

<<***eXtension***>> *(p. 199)* Asynchronous publishing settings for the Discovery **DDSPublisher** (p. 1346) and all entities that are created by it.

6.29 DDS_DiscoveryQosPolicy Struct Reference

Configures the mechanism used by the middleware to automatically discover and connect with new remote applications.

Public Attributes

- struct **DDS_StringSeq enabled_transports**

  The transports available for use by the Discovery mechanism.

- struct **DDS_StringSeq initial_peers**

  Determines the initial list of peers that will be contacted by the Discovery mechanism to send announcements about the presence of this participant.

- struct **DDS_StringSeq multicast_receive_addresses**

  Specifies the multicast group addresses on which discovery-related **meta-traffic** *can be received by the DomainParticipant.*

- **DDS_Long metatraffic_transport_priority**

  The transport priority to use for the Discovery meta-traffic.

- **DDS_Boolean accept_unknown_peers**

  Whether to accept a new participant that is not in the initial peers list.

6.29.1 Detailed Description

Configures the mechanism used by the middleware to automatically discover and connect with new remote applications.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.29.2 Usage

This QoS policy identifies where on the network this application can *potentially* discover other applications with which to communicate.

The middleware will periodically send network packets to these locations, announcing itself to any remote applications that may be present, and will listen for announcements from those applications.

This QoS policy is an extension to the DDS standard.

See also:

NDDS_DISCOVERY_PEERS (p. 388)
DDS_DiscoveryConfigQosPolicy (p. 573)

6.29.3 Member Data Documentation

6.29.3.1 struct DDS_StringSeq DDS_DiscoveryQosPolicy::enabled_transports [read]

The transports available for use by the Discovery mechanism.

Only these transports can be used by the discovery mechanism to send meta-traffic via the builtin endpoints (built-in **DDSDataReader** (p. 1087) and **DDSDataWriter** (p. 1113)).

Also determines the unicast addresses on which the Discovery mechanism will listen for meta-traffic. These along with the `domain_id` and `participant_id` determine the unicast locators on which the Discovery mechanism can receive meta-data.

The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

Alias names for the builtin transports are defined in **TRANSPORT_BUILTIN** (p. 396).

[**default**] Empty sequence. All the transports available to the DomainParticipant are available for use by the Discovery mechanism.

[**range**] Sequence of non-null,non-empty strings.

6.29.3.2 struct DDS_StringSeq DDS_DiscoveryQosPolicy::initial_peers [read]

Determines the initial list of peers that will be contacted by the Discovery mechanism to send announcements about the presence of this participant.

If there is a remote peer **DDSDomainParticipant** (p. 1139) such as is described in this list, it will become aware of this participant and will engage in

the Discovery protocol to exchange meta-data with this participant.

Each element of this list must be a `peer` descriptor in the proper format (see **Peer Descriptor Format** (p. 389)).

[default] See **NDDS_DISCOVERY_PEERS** (p. 388)

[range] Sequence of arbitrary length.

See also:

Peer Descriptor Format (p. 389)
DDSDomainParticipant::add_peer() (p. 1199)

6.29.3.3 struct DDS_StringSeq DDS_DiscoveryQosPolicy::multicast_receive_addresses [read]

Specifies the multicast group addresses on which discovery-related **meta-traffic** can be received by the DomainParticipant.

The multicast group addresses on which the Discovery mechanism will listen for meta-traffic.

Each element of this list must be a valid multicast address (IPv4 or IPv6) in the proper format (see **Address Format** (p. 390)).

The `domain_id` determines the multicast port on which the Discovery mechanism can receive meta-data.

If NDDS_DISCOVERY_PEERS does *not* contain a multicast address, then the string sequence **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584) is cleared and the RTI discovery process will not listen for discovery messages via multicast.

If NDDS_DISCOVERY_PEERS contains one or more multicast addresses, the addresses will be stored in **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584), starting at element 0. They will be stored in the order they appear NDDS_DISCOVERY_PEERS.

Note: Currently, RTI Connext will only listen for discovery traffic on the first multicast address (element 0) in **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584).

[default] See **NDDS_DISCOVERY_PEERS** (p. 388)

[range] Sequence of length [0,1], whose elements are multicast addresses. Currently only the first multicast address (if any) is used. The rest are ignored.

See also:

Address Format (p. 390)

6.29.3.4 DDS_Long DDS_DiscoveryQosPolicy::metatraffic_transport_priority

The transport priority to use for the Discovery meta-traffic.

The discovery metatraffic will be sent by the built-in **DDSDataWriter** (p. 1113) using this transport priority.

[**default**] 0

6.29.3.5 DDS_Boolean DDS_DiscoveryQosPolicy::accept_unknown_peers

Whether to accept a new participant that is not in the initial peers list.

If **DDS_BOOLEAN_FALSE** (p. 299), the participant will only communicate with those in the initial peers list and those added via **DDSDomainParticipant::add_peer()** (p. 1199).

If **DDS_BOOLEAN_TRUE** (p. 298), the participant will also communicate with all discovered remote participants.

Note: If accept_unknown_peers is **DDS_BOOLEAN_FALSE** (p. 299) and shared memory is disabled, applications on the same node will *not* communicate if only 'localhost' is specified in the peers list. If shared memory is disabled or 'shmem://' is not specified in the peers list, to communicate with other applications on the same node through the loopback interface, you must put the actual node address or hostname in **NDDS_DISCOVERY_PEERS** (p. 388).

[**default**] **DDS_BOOLEAN_TRUE** (p. 298)

6.30 DDS_DomainParticipantFactoryQos Struct Reference

QoS policies supported by a **DDSDomainParticipantFactory** (p. 1216).

Public Attributes

- struct **DDS_EntityFactoryQosPolicy entity_factory**

  Entity factory policy, ***ENTITY_FACTORY*** *(p. 377).*

- struct **DDS_SystemResourceLimitsQosPolicy resource_limits**

  <<**eXtension**>> (p. 199) *System resource limits,* ***SYSTEM_RESOURCE_LIMITS*** *(p. 415).*

- struct **DDS_ProfileQosPolicy profile**

  <<**eXtension**>> (p. 199) *Qos profile policy,* ***PROFILE*** *(p. 446).*

- struct **DDS_LoggingQosPolicy logging**

  <<**eXtension**>> (p. 199) *Logging qos policy,* ***LOGGING*** *(p. 450).*

6.30.1 Detailed Description

QoS policies supported by a **DDSDomainParticipantFactory** (p. 1216).

Entity:

DDSDomainParticipantFactory (p. 1216)

See also:

QoS Policies (p. 331) and allowed ranges within each Qos.

6.30.2 Member Data Documentation

6.30.2.1 struct DDS_EntityFactoryQosPolicy DDS_DomainParticipantFactoryQos::entity_factory [read]

Entity factory policy, **ENTITY_FACTORY** (p. 377).

6.30.2.2 struct DDS_SystemResourceLimitsQosPolicy DDS_DomainParticipantFactoryQos::resource_limits [read]

<<*eXtension*>> *(p. 199)* System resource limits, **SYSTEM_RESOURCE_LIMITS** (p. 415).

6.30.2.3 struct DDS_ProfileQosPolicy DDS_DomainParticipantFactoryQos::profile [read]

<<*eXtension*>> *(p. 199)* Qos profile policy, **PROFILE** (p. 446).

6.30.2.4 struct DDS_LoggingQosPolicy DDS_DomainParticipantFactoryQos::logging [read]

<<*eXtension*>> *(p. 199)* Logging qos policy, **LOGGING** (p. 450).

6.31 DDS_DomainParticipantQos Struct Reference

QoS policies supported by a **DDSDomainParticipant** (p. 1139) entity.

Public Attributes

- struct **DDS_UserDataQosPolicy user_data**

  User data policy, ***USER_DATA*** *(p. 345).*

- struct **DDS_EntityFactoryQosPolicy entity_factory**

  Entity factory policy, ***ENTITY_FACTORY*** *(p. 377).*

- struct **DDS_WireProtocolQosPolicy wire_protocol**

  <<**eXtension**>> (p. 199) *Wire Protocol policy,* ***WIRE_PROTOCOL*** *(p. 400).*

- struct **DDS_TransportBuiltinQosPolicy transport_builtin**

  <<**eXtension**>> (p. 199) *Transport Builtin policy,* ***TRANSPORT_BUILTIN*** *(p. 396).*

- struct **DDS_TransportUnicastQosPolicy default_unicast**

  <<**eXtension**>> (p. 199) *Default Unicast Transport policy,* ***TRANSPORT_UNICAST*** *(p. 383).*

- struct **DDS_DiscoveryQosPolicy discovery**

  <<**eXtension**>> (p. 199) *Discovery policy,* ***DISCOVERY*** *(p. 387).*

- struct **DDS_DomainParticipantResourceLimitsQosPolicy resource_limits**

  <<**eXtension**>> (p. 199) *Domain participant resource limits policy,* ***DOMAIN_PARTICIPANT_RESOURCE_LIMITS*** *(p. 416).*

- struct **DDS_EventQosPolicy event**

  <<**eXtension**>> (p. 199) *Event policy,* ***EVENT*** *(p. 417).*

- struct **DDS_ReceiverPoolQosPolicy receiver_pool**

  <<**eXtension**>> (p. 199) *Receiver pool policy,* ***RECEIVER_POOL*** *(p. 419).*

- struct **DDS_DatabaseQosPolicy database**

  <<**eXtension**>> (p. 199) *Database policy,* ***DATABASE*** *(p. 418).*

- struct **DDS_DiscoveryConfigQosPolicy discovery_config**

  <<**eXtension**>> (p. 199) *Discovery config policy,* ***DISCOVERY_CONFIG*** *(p. 423).*

- struct **DDS_PropertyQosPolicy property**

  <<**eXtension**>> (p. 199) *Property policy,* ***PROPERTY*** *(p. 436).*

- struct **DDS_EntityNameQosPolicy participant_name**

  <<**eXtension**>> (p. 199) *The participant name.* ***ENTITY_NAME*** *(p. 445)*

- struct **DDS_TransportMulticastMappingQosPolicy multicast_mapping**

  <<**eXtension**>> (p. 199) *The multicast mapping policy.* ***TRANSPORT_MULTICAST_MAPPING*** *(p. 386)*

- struct **DDS_TypeSupportQosPolicy type_support**

  <<**eXtension**>> (p. 199) *Type support data,* ***TYPESUPPORT*** *(p. 429).*

6.31.1 Detailed Description

QoS policies supported by a **DDSDomainParticipant** (p. 1139) entity.

Certain members must be set in a consistent manner:

Length of **DDS_DomainParticipantQos::user_data** (p. 590) .value <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .participant_user_data_max_length

For **DDS_DomainParticipantQos::discovery_config** (p. 591) .publication_writer

high_watermark <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .local_writer_allocation .max_count heartbeats_per_max_samples <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .local_writer_allocation.max_count

For **DDS_DomainParticipantQos::discovery_config** (p. 591) .suscription_writer

high_watermark <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .local_reader_allocation.max_count heartbeats_per_max_samples <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .local_reader_allocation.max_count

If any of the above are not true, **DDSDomainParticipant::set_qos** (p. 1197) and **DDSDomainParticipant::set_qos_with_profile** (p. 1198) and **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315), and **DDSDomainParticipantFactory::create_participant** (p. 1233) will fail.

Entity:

DDSDomainParticipant (p. 1139)

See also:

QoS Policies (p. 331) and allowed ranges within each Qos.
NDDS_DISCOVERY_PEERS (p. 388)

6.31.2 Member Data Documentation

6.31.2.1 struct DDS_UserDataQosPolicy DDS_DomainParticipantQos::user_data [read]

User data policy, **USER_DATA** (p. 345).

6.31.2.2 struct DDS_EntityFactoryQosPolicy DDS_DomainParticipantQos::entity_factory [read]

Entity factory policy, **ENTITY_FACTORY** (p. 377).

6.31.2.3 struct DDS_WireProtocolQosPolicy DDS_DomainParticipantQos::wire_protocol [read]

<<***eXtension***>> *(p. 199)* Wire Protocol policy, **WIRE_PROTOCOL** (p. 400).

The wire protocol (RTPS) attributes associated with the participant.

6.31.2.4 struct DDS_TransportBuiltinQosPolicy DDS_DomainParticipantQos::transport_builtin [read]

<<***eXtension***>> *(p. 199)* Transport Builtin policy, **TRANSPORT_BUILTIN** (p. 396).

6.31.2.5 struct DDS_TransportUnicastQosPolicy DDS_DomainParticipantQos::default_unicast [read]

<<*eXtension*>> *(p. 199)* Default Unicast Transport policy, **TRANSPORT_UNICAST** (p. 383).

6.31.2.6 struct DDS_DiscoveryQosPolicy DDS_DomainParticipantQos::discovery [read]

<<*eXtension*>> *(p. 199)* Discovery policy, **DISCOVERY** (p. 387).

6.31.2.7 struct DDS_DomainParticipantResourceLimitsQosPolicy DDS_DomainParticipantQos::resource_limits [read]

<<*eXtension*>> *(p. 199)* Domain participant resource limits policy, **DOMAIN_PARTICIPANT_RESOURCE_LIMITS** (p. 416).

6.31.2.8 struct DDS_EventQosPolicy DDS_DomainParticipantQos::event [read]

<<*eXtension*>> *(p. 199)* Event policy, **EVENT** (p. 417).

6.31.2.9 struct DDS_ReceiverPoolQosPolicy DDS_DomainParticipantQos::receiver_pool [read]

<<*eXtension*>> *(p. 199)* Receiver pool policy, **RECEIVER_POOL** (p. 419).

6.31.2.10 struct DDS_DatabaseQosPolicy DDS_DomainParticipantQos::database [read]

<<*eXtension*>> *(p. 199)* Database policy, **DATABASE** (p. 418).

6.31.2.11 struct DDS_DiscoveryConfigQosPolicy DDS_DomainParticipantQos::discovery_config [read]

<<*eXtension*>> *(p. 199)* Discovery config policy, **DISCOVERY_CONFIG** (p. 423).

6.31.2.12 struct DDS_PropertyQosPolicy DDS_DomainParticipantQos::property [read]

<<*eXtension*>> *(p. 199)* Property policy, **PROPERTY** (p. 436).

6.31.2.13 struct DDS_EntityNameQosPolicy DDS_DomainParticipantQos::participant_name [read]

<<*eXtension*>> *(p. 199)* The participant name. **ENTITY_NAME** (p. 445)

6.31.2.14 struct DDS_TransportMulticastMappingQosPolicy DDS_DomainParticipantQos::multicast_mapping [read]

<<*eXtension*>> *(p. 199)* The multicast mapping policy. **TRANSPORT_MULTICAST_MAPPING** (p. 386)

6.31.2.15 struct DDS_TypeSupportQosPolicy DDS_DomainParticipantQos::type_support [read]

<<*eXtension*>> *(p. 199)* Type support data, **TYPESUPPORT** (p. 429).

Optional value that is passed to a type plugin's on_participant_attached function.

6.32 DDS_DomainParticipantResourceLimitsQosPolicy Struct Reference

Various settings that configure how a **DDSDomainParticipant** (p. 1139) allocates and uses physical memory for internal resources, including the maximum sizes of various properties.

Public Attributes

- struct **DDS_AllocationSettings_t local_writer_allocation**

  Allocation settings applied to local DataWriters.

- struct **DDS_AllocationSettings_t local_reader_allocation**

  Allocation settings applied to local DataReaders.

- struct **DDS_AllocationSettings_t local_publisher_allocation**

  Allocation settings applied to local Publisher.

- struct **DDS_AllocationSettings_t local_subscriber_allocation**

  Allocation settings applied to local Subscriber.

- struct **DDS_AllocationSettings_t local_topic_allocation**

  Allocation settings applied to local Topic.

- struct **DDS_AllocationSettings_t remote_writer_allocation**

  Allocation settings applied to remote DataWriters.

- struct **DDS_AllocationSettings_t remote_reader_allocation**

  Allocation settings applied to remote DataReaders.

- struct **DDS_AllocationSettings_t remote_participant_allocation**

  Allocation settings applied to remote DomainParticipants.

- struct **DDS_AllocationSettings_t matching_writer_reader_pair_allocation**

  Allocation settings applied to matching local writer and remote/local reader pairs.

- struct **DDS_AllocationSettings_t matching_reader_writer_pair_allocation**

  Allocation settings applied to matching local reader and remote/local writer pairs.

- struct **DDS_AllocationSettings_t ignored_entity_allocation**

  Allocation settings applied to ignored entities.

- struct **DDS_AllocationSettings_t content_filtered_topic_allocation**

  Allocation settings applied to content filtered topic.

- struct **DDS_AllocationSettings_t content_filter_allocation**

  Allocation settings applied to content filter.

- struct **DDS_AllocationSettings_t read_condition_allocation**

  Allocation settings applied to read condition pool.

- struct **DDS_AllocationSettings_t query_condition_allocation**

  Allocation settings applied to query condition pool.

- struct **DDS_AllocationSettings_t outstanding_asynchronous_sample_allocation**

  *Allocation settings applied to the maximum number of samples (from all **DDSDataWriter** (p. 1113)) waiting to be asynchronously written.*

- struct **DDS_AllocationSettings_t flow_controller_allocation**

  Allocation settings applied to flow controllers.

- **DDS_Long local_writer_hash_buckets**

  Hash_Buckets settings applied to local DataWriters.

- **DDS_Long local_reader_hash_buckets**

  Number of hash buckets for local DataReaders.

- **DDS_Long local_publisher_hash_buckets**

  Number of hash buckets for local Publisher.

- **DDS_Long local_subscriber_hash_buckets**

  Number of hash buckets for local Subscriber.

- **DDS_Long local_topic_hash_buckets**

  Number of hash buckets for local Topic.

- **DDS_Long remote_writer_hash_buckets**

  Number of hash buckets for remote DataWriters.

- **DDS_Long remote_reader_hash_buckets**

  Number of hash buckets for remote DataReaders.

- **DDS_Long remote_participant_hash_buckets**

  Number of hash buckets for remote DomainParticipants.

- **DDS_Long matching_writer_reader_pair_hash_buckets**

  Number of hash buckets for matching local writer and remote/local reader pairs.

- **DDS_Long matching_reader_writer_pair_hash_buckets**

  Number of hash buckets for matching local reader and remote/local writer pairs.

- **DDS_Long ignored_entity_hash_buckets**

  Number of hash buckets for ignored entities.

- **DDS_Long content_filtered_topic_hash_buckets**

  Number of hash buckets for content filtered topics.

- **DDS_Long content_filter_hash_buckets**

  Number of hash buckets for content filters.

- **DDS_Long flow_controller_hash_buckets**

  Number of hash buckets for flow controllers.

- **DDS_Long max_gather_destinations**

  Maximum number of destinations per RTI Connext send.

- **DDS_Long participant_user_data_max_length**

  Maximum length of user data in ***DDS_DomainParticipantQos*** *(p. 588) and* ***DDS_ParticipantBuiltinTopicData*** *(p. 816).*

- **DDS_Long topic_data_max_length**

  Maximum length of topic data in ***DDS_TopicQos*** *(p. 965),* ***DDS_TopicBuiltinTopicData*** *(p. 958),* ***DDS_PublicationBuiltinTopicData*** *(p. 839) and* ***DDS_SubscriptionBuiltinTopicData*** *(p. 936).*

- **DDS_Long publisher_group_data_max_length**

  Maximum length of group data in ***DDS_PublisherQos*** *(p. 851) and* ***DDS_PublicationBuiltinTopicData*** *(p. 839).*

- **DDS_Long subscriber_group_data_max_length**

*Maximum length of group data in **DDS_SubscriberQos** (p. 934) and **DDS_SubscriptionBuiltinTopicData** (p. 936).*

- **DDS_Long writer_user_data_max_length**

  *Maximum length of user data in **DDS_DataWriterQos** (p. 553) and **DDS_PublicationBuiltinTopicData** (p. 839).*

- **DDS_Long reader_user_data_max_length**

  *Maximum length of user data in **DDS_DataReaderQos** (p. 515) and **DDS_SubscriptionBuiltinTopicData** (p. 936).*

- **DDS_Long max_partitions**

  *Maximum number of partition name strings allowable in a **DDS_PartitionQosPolicy** (p. 820).*

- **DDS_Long max_partition_cumulative_characters**

  *Maximum number of combined characters allowable in all partition names in a **DDS_PartitionQosPolicy** (p. 820).*

- **DDS_Long type_code_max_serialized_length**

  Maximum size of serialized string for type code.

- **DDS_Long type_object_max_serialized_length**
- **DDS_Long serialized_type_object_dynamic_allocation_threshold**
- **DDS_Long type_object_max_deserialized_length**
- **DDS_Long deserialized_type_object_dynamic_allocation_threshold**
- **DDS_Long contentfilter_property_max_length**

  This field is the maximum length of all data related to a Content-filtered topic.

- **DDS_Long channel_seq_max_length**

  *Maximum number of channels that can be specified in **DDS_MultiChannelQosPolicy** (p. 796) for MultiChannel DataWriters.*

- **DDS_Long channel_filter_expression_max_length**

  *Maximum length of a channel **DDS_ChannelSettings_t::filter_expression** (p. 486) in a MultiChannel DataWriter.*

- **DDS_Long participant_property_list_max_length**

  *Maximum number of properties associated with the **DDSDomainParticipant** (p. 1139).*

- **DDS_Long participant_property_string_max_length**

*Maximum string length of the properties associated with the **DDSDomainParticipant** (p. 1139).*

- **DDS_Long writer_property_list_max_length**

  *Maximum number of properties associated with a **DDSDataWriter** (p. 1113).*

- **DDS_Long writer_property_string_max_length**

  *Maximum string length of the properties associated with a **DDSDataWriter** (p. 1113).*

- **DDS_Long reader_property_list_max_length**

  *Maximum number of properties associated with a **DDSDataReader** (p. 1087).*

- **DDS_Long reader_property_string_max_length**

  *Maximum string length of the properties associated with a **DDSDataReader** (p. 1087).*

- **DDS_Long max_endpoint_groups**

  *Maximum number of **DDS_EndpointGroup_t** (p. 731) allowable in a **DDS_AvailabilityQosPolicy** (p. 471).*

- **DDS_Long max_endpoint_group_cumulative_characters**

  *Maximum number of combined role_name characters allowable in all **DDS_EndpointGroup_t** (p. 731) in a **DDS_AvailabilityQosPolicy** (p. 471).*

6.32.1 Detailed Description

Various settings that configure how a **DDSDomainParticipant** (p. 1139) allocates and uses physical memory for internal resources, including the maximum sizes of various properties.

This QoS policy sets maximum size limits on variable-length parameters used by the participant and its contained Entities. It also controls the initial and maximum sizes of data structures used by the participant to store information about locally-created and remotely-discovered entities (such as DataWriters/DataReaders), as well as parameters used by the internal database to size the hash tables it uses.

By default, a **DDSDomainParticipant** (p. 1139) is allowed to dynamically allocate memory as needed as users create local Entities such as **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) objects or as the participant discovers new applications. By setting fixed values for the maximum

parameters in this QoS policy, you can bound the memory that can be allocated by a **DDSDomainParticipant** (p. 1139). In addition, by setting the initial values to the maximum values, you can prevent DomainParticipants from allocating memory after the initialization period.

The maximum sizes of different variable-length parameters such as the number of partitions that can be stored in the **DDS_PartitionQosPolicy** (p. 820), the maximum length of data store in the **DDS_UserDataQosPolicy** (p. 1048) and **DDS_GroupDataQosPolicy** (p. 755), and many others can be changed from their defaults using this QoS policy. However, it is important that all DomainParticipants that need to communicate with each other use the *same set* of maximum values. Otherwise, when these parameters are propagated from one **DDSDomainParticipant** (p. 1139) to another, a **DDSDomainParticipant** (p. 1139) with a smaller maximum length may reject the parameter, resulting in an error.

An important parameter in this QoS policy that is often changed by users is **DDS_DomainParticipantResourceLimitsQosPolicy::type_code_max_serialized_length** (p. 608).

This QoS policy is an extension to the DDS standard.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.32.2 Member Data Documentation

6.32.2.1 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::local_writer_allocation [read]

Allocation settings applied to local DataWriters.

[default] initial_count = 16; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.2 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::local_reader_allocation [read]

Allocation settings applied to local DataReaders.

[**default**] initial_count = 16; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.3 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::local_publisher_allocation [read]

Allocation settings applied to local Publisher.

[**default**] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.4 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::local_subscriber_allocation [read]

Allocation settings applied to local Subscriber.

[**default**] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.5 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::local_topic_allocation [read]

Allocation settings applied to local Topic.

[**default**] initial_count = 16; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.6 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::remote_writer_allocation [read]

Allocation settings applied to remote DataWriters.

Remote DataWriters include all DataWriters, both local and remote.

[default] initial_count = 64; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.7 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::remote_reader_allocation [read]

Allocation settings applied to remote DataReaders.

Remote DataReaders include all DataReaders, both local and remote.

[default] initial_count = 64; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.8 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::remote_participant_allocation [read]

Allocation settings applied to remote DomainParticipants.

Remote DomainParticipants include all DomainParticipants, both local and remote.

[default] initial_count = 16; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.9 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::matching_writer_reader_pair_allocation [read]

Allocation settings applied to matching local writer and remote/local reader pairs.

[default] initial_count = 32; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.10 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::matching_reader_writer_pair_allocation [read]

Allocation settings applied to matching local reader and remote/local writer pairs.

[**default**] initial_count = 32; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.11 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::ignored_entity_allocation [read]

Allocation settings applied to ignored entities.

[**default**] initial_count = 8; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.12 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::content_filtered_topic_allocation [read]

Allocation settings applied to content filtered topic.

[**default**] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.13 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::content_filter_allocation [read]

Allocation settings applied to content filter.

[**default**] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371); incremental_count = -1

[**range**] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.14 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::read_condition_allocation [read]

Allocation settings applied to read condition pool.

[default] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371), incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.15 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::query_condition_allocation [read]

Allocation settings applied to query condition pool.

[default] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371), incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.16 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::outstanding_asynchronous_sample_allocation [read]

Allocation settings applied to the maximum number of samples (from all **DDS-DataWriter** (p. 1113)) waiting to be asynchronously written.

[default] initial_count = 64; max_count = **DDS_LENGTH_UNLIMITED** (p. 371), incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.17 struct DDS_AllocationSettings_t DDS_DomainParticipantResourceLimitsQosPolicy::flow_controller_allocation [read]

Allocation settings applied to flow controllers.

[default] initial_count = 4; max_count = **DDS_LENGTH_UNLIMITED** (p. 371), incremental_count = -1

[range] See allowed ranges in struct **DDS_AllocationSettings_t** (p. 464)

6.32.2.18 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::local_writer_hash_buckets

Hash_Buckets settings applied to local DataWriters.

[default] 4

[range] [1, 10000]

6.32.2.19 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::local_reader_hash_buckets

Number of hash buckets for local DataReaders.

[default] 4

[range] [1, 10000]

6.32.2.20 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::local_publisher_hash_buckets

Number of hash buckets for local Publisher.

[default] 1

[range] [1, 10000]

6.32.2.21 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::local_subscriber_hash_buckets

Number of hash buckets for local Subscriber.

[default] 1

[range] [1, 10000]

6.32.2.22 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::local_topic_hash_buckets

Number of hash buckets for local Topic.

[default] 4

[range] [1, 10000]

6.32.2.23 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::remote_writer_hash_buckets

Number of hash buckets for remote DataWriters.

Remote DataWriters include all DataWriters, both local and remote.

[default] 16

[range] [1, 10000]

6.32.2.24 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::remote_reader_hash_buckets

Number of hash buckets for remote DataReaders.

Remote DataReaders include all DataReaders, both local and remote.

[default] 16

[range] [1, 10000]

6.32.2.25 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::remote_participant_hash_buckets

Number of hash buckets for remote DomainParticipants.

Remote DomainParticipants include all DomainParticipants, both local and remote.

[default] 4

[range] [1, 10000]

6.32.2.26 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::matching_writer_reader_pair_hash_buckets

Number of hash buckets for matching local writer and remote/local reader pairs.

[default] 32

[range] [1, 10000]

6.32.2.27 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::matching_reader_writer_pair_hash_buckets

Number of hash buckets for matching local reader and remote/local writer pairs.

[default] 32

[range] [1, 10000]

6.32.2.28 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::ignored_entity_hash_buckets

Number of hash buckets for ignored entities.

[default] 1

[range] [1, 10000]

6.32.2.29 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::content_filtered_topic_hash_buckets

Number of hash buckets for content filtered topics.

[default] 1

[range] [1, 10000]

6.32.2.30 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::content_filter_hash_buckets

Number of hash buckets for content filters.

[default] 1

[range] [1, 10000]

6.32.2.31 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::flow_controller_hash_buckets

Number of hash buckets for flow controllers.

[default] 1

[range] [1, 10000]

6.32.2.32 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::max_gather_destinations

Maximum number of destinations per RTI Connext send.

When RTI Connext sends out a message, it has the capability to send to multiple destinations to be more efficient. The maximum number of destinations per RTI Connext send is specified by max_gather_destinations.

[default] 8

[range] [4, 1 million]

6.32.2.33 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::participant_user_data_max_length

Maximum length of user data in **DDS_DomainParticipantQos** (p. 588) and **DDS_ParticipantBuiltinTopicData** (p. 816).

[default] 256

[range] [0,0x7fffffff]

6.32.2.34 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::topic_data_max_length

Maximum length of topic data in **DDS_TopicQos** (p. 965), **DDS_TopicBuiltinTopicData** (p. 958), **DDS_PublicationBuiltinTopicData** (p. 839) and **DDS_SubscriptionBuiltinTopicData** (p. 936).

[default] 256

[range] [0,0x7fffffff]

6.32.2.35 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::publisher_group_data_max_length

Maximum length of group data in **DDS_PublisherQos** (p. 851) and **DDS_PublicationBuiltinTopicData** (p. 839).

[default] 256

[**range**] [0,0x7fffffff]

6.32.2.36 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::subscriber_group_data_max_length

Maximum length of group data in **DDS_SubscriberQos** (p. 934) and **DDS_SubscriptionBuiltinTopicData** (p. 936).

[**default**] 256

[**range**] [0,0x7fffffff]

6.32.2.37 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::writer_user_data_max_length

Maximum length of user data in **DDS_DataWriterQos** (p. 553) and **DDS_PublicationBuiltinTopicData** (p. 839).

[**default**] 256

[**range**] [0,0x7fffffff]

6.32.2.38 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::reader_user_data_max_length

Maximum length of user data in **DDS_DataReaderQos** (p. 515) and **DDS_SubscriptionBuiltinTopicData** (p. 936).

[**default**] 256

[**range**] [0,0x7fffffff]

6.32.2.39 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::max_partitions

Maximum number of partition name strings allowable in a **DDS_PartitionQosPolicy** (p. 820).

This setting is made on a per DomainParticipant basis; it cannot be set individually on a per Publisher/Subscriber basis. However, the limit is enforced and applies per Publisher/Subscriber.

This value cannot exceed 64.

[default] 64

[range] [0,64]

6.32.2.40 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::max_partition_cumulative_characters

Maximum number of combined characters allowable in all partition names in a **DDS_PartitionQosPolicy** (p. 820).

The maximum number of combined characters should account for a terminating NULL ('\0') character for each partition name string.

This setting is made on a per DomainParticipant basis; it cannot be set individually on a per Publisher/Subscriber basis. However, the limit is enforced and applies per Publisher/Subscriber.

This value cannot exceed 256.

[default] 256

[range] [0,256]

6.32.2.41 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::type_code_max_serialized_length

Maximum size of serialized string for type code.

This parameter limits the size of the type code that a **DDSDomainParticipant** (p. 1139) is able to store and propagate for user data types. Type codes can be used by external applications to understand user data types without having the data type predefined in compiled form. However, since type codes 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 type codes larger than the default maximum of 2048 bytes. So it is common for users to set this parameter to a larger value. However, as with all parameters in this QoS policy defining maximum sizes for variable-length elements, all DomainParticipants in the same domain should use the same value for this parameter.

[default] 2048

[range] [0,0xffff]

6.32.2.42 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::type_object_max_serialized_length

6.32.2.43 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::serialized_type_object_dynamic_allocation_threshold

6.32.2.44 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::type_object_max_deserialized_length

6.32.2.45 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::deserialized_type_object_dynamic_allocation_threshold

6.32.2.46 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::contentfilter_property_max_length

This field is the maximum length of all data related to a Content-filtered topic.

This is the sum of the length of the content filter name, the length of the related topic name, the length of the filter expression, the length of the filter parameters, and the length of the filter name. The maximum number of combined characters should account for a terminating NULL ('\0') character for each string.

[**default**] 256

[**range**] [0,0xffff]

6.32.2.47 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::channel_seq_max_length

Maximum number of channels that can be specified in **DDS_MultiChannelQosPolicy** (p. 796) for MultiChannel DataWriters.

[**default**] 32

[**range**] [0,0xffff]

6.32.2.48 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::channel_filter_expression_max_length

Maximum length of a channel **DDS_ChannelSettings_t::filter_expression** (p. 486) in a MultiChannel DataWriter.

The length should account for a terminating NULL ('\0') character.

[default] 256

[range] [0,0xffff]

6.32.2.49 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::participant_property_list_max_length

Maximum number of properties associated with the **DDSDomainParticipant** (p. 1139).

[default] 32

[range] [0,0xffff]

6.32.2.50 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::participant_property_string_max_length

Maximum string length of the properties associated with the **DDSDomainParticipant** (p. 1139).

The string length is defined as the cumulative length in bytes, including the null terminating characters, of all the pair (name,value) associated with the **DDSDomainParticipant** (p. 1139) properties.

[default] 1024

[range] [0,0xffff]

6.32.2.51 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::writer_property_list_max_length

Maximum number of properties associated with a **DDSDataWriter** (p. 1113).

[range] [0,0xffff]

[default] 32

6.32.2.52 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::writer_property_string_max_length

Maximum string length of the properties associated with a **DDSDataWriter** (p. 1113).

The string length is defined as the cumulative length in bytes, including the null terminating characters, of all the pair (name,value) associated with the **DDSDataWriter** (p. 1113) properties.

[default] 1024

[range] [0,0xffff]

6.32.2.53 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::reader_property_list_max_length

Maximum number of properties associated with a **DDSDataReader** (p. 1087).

[default] 32

[range] [0,0xffff]

6.32.2.54 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::reader_property_string_max_length

Maximum string length of the properties associated with a **DDSDataReader** (p. 1087).

The string length is defined as the cumulative length in bytes, including the null terminating characters, of all the pair (name,value) associated with a **DDSDataReader** (p. 1087) properties.

[default] 1024

[range] [0,0xffff]

6.32.2.55 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::max_endpoint_groups

Maximum number of **DDS_EndpointGroup_t** (p. 731) allowable in a **DDS_AvailabilityQosPolicy** (p. 471).

[default] 32

[range] [0,65535]

6.32.2.56 DDS_Long DDS_DomainParticipantResourceLimitsQosPolicy::max_endpoint_group_cumulative_characters

Maximum number of combined role_name characters allowable in all **DDS_EndpointGroup_t** (p. 731) in a **DDS_AvailabilityQosPolicy** (p. 471).

The maximum number of combined characters should account for a terminating NULL character for each role_name string.

[default] 1024

[range] [0,65535]

6.33 DDS_DoubleSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_Double** (p. 300) >.

6.33.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_Double** (p. 300) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Double (p. 300)
FooSeq (p. 1494)

6.34 DDS_DurabilityQosPolicy Struct Reference

This QoS policy specifies whether or not RTI Connext will store and deliver previously published data samples to new **DDSDataReader** (p. 1087) entities that join the network later.

Public Attributes

- **DDS_DurabilityQosPolicyKind kind**

  The kind of durability.

- **DDS_Boolean direct_communication**

  <<**eXtension**>> (p. 199) *Indicates whether or not a TRANSIENT or PERSISTENT **DDSDataReader** (p. 1087) should receive samples directly from a TRANSIENT or PERSISTENT **DDSDataWriter** (p. 1113)*

6.34.1 Detailed Description

This QoS policy specifies whether or not RTI Connext will store and deliver previously published data samples to new **DDSDataReader** (p. 1087) entities that join the network later.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

See also:

DURABILITY_SERVICE (p. 370)

6.34.2 Usage

It is possible for a **DDSDataWriter** (p. 1113) to start publishing data before all (or any) **DDSDataReader** (p. 1087) entities have joined the network.

Moreover, a **DDSDataReader** (p. 1087) that joins the network after some data has been written could potentially be interested in accessing the most current values of the data, as well as potentially some history.

This policy makes it possible for a late-joining **DDSDataReader** (p. 1087) to obtain previously published samples.

By helping to ensure that DataReaders get all data that was sent by DataWriters, regardless of when it was sent, using this QoS policy can increase system tolerance to failure conditions.

Note that although related, this does not strictly control what data RTI Connext will maintain internally. That is, RTI Connext may choose to maintain some data for its own purposes (e.g., flow control) and yet not make it available to late-joining readers if the **DURABILITY** (p. 348) policy is set to **DDS_VOLATILE_DURABILITY_QOS** (p. 349).

6.34.2.1 Transient and Persistent Durability

For the purpose of implementing the DURABILITY QoS kind TRANSIENT or PERSISTENT, RTI Connext behaves *as if* for each Topic that has **DDS_DurabilityQosPolicy::kind** (p. 617) of **DDS_TRANSIENT_DURABILITY_QOS** (p. 349) or **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) there is a corresponding "built-in" **DDSDataReader** (p. 1087) and **DDSDataWriter** (p. 1113) configured with the same DURABILITY kind. In other words, it is *as if* somewhere in the system, independent of the original **DDSDataWriter** (p. 1113), there is a built-in durable **DDSDataReader** (p. 1087) subscribing to that Topic and a built-in durable DataWriter republishing it as needed for the new subscribers that join the system. This functionality is provided by the *RTI Persistence Service*.

The Persistence Service can configure itself based on the QoS of your application's **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) entities. For each transient or persistent **DDSTopic** (p. 1419), the built-in fictitious Persistence Service **DDSDataReader** (p. 1087) and **DDSDataWriter** (p. 1113) have their QoS configured from the QoS of your application's **DDSDataWriter** (p. 1113) and **DDSDataReader** (p. 1087) entities that communicate on that **DDSTopic** (p. 1419).

For a given **DDSTopic** (p. 1419), the usual request/offered semantics apply to the matching between any **DDSDataWriter** (p. 1113) in the domain that writes the **DDSTopic** (p. 1419) and the built-in transient/persistent **DDSDataReader** (p. 1087) for that **DDSTopic** (p. 1419); similarly for the built-

in transient/persistent **DDSDataWriter** (p. 1113) for a **DDSTopic** (p. 1419) and any **DDSDataReader** (p. 1087) for the **DDSTopic** (p. 1419). As a consequence, a **DDSDataWriter** (p. 1113) that has an incompatible QoS will not send its data to the *RTI Persistence Service*, and a **DDSDataReader** (p. 1087) that has an incompatible QoS will not get data from it.

Incompatibilities between local **DDSDataReader** (p. 1087) and **DDSDataWriter** (p. 1113) entities and the corresponding fictitious built-in transient/persistent entities cause the **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323) and **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323) to change and the corresponding Listener invocations and/or signaling of **DDSCondition** (p. 1075) objects as they would with your application's own entities.

The value of **DDS_DurabilityServiceQosPolicy::service_cleanup_delay** (p. 619) controls when *RTI Persistence Service* is able to remove all information regarding a data instances.

Information on a data instance is maintained until the following conditions are met:

1. The instance has been explicitly disposed (instance_state = NOT_ALIVE_DISPOSED),

and

2. While in the NOT_ALIVE_DISPOSED state, the system detects that there are no more 'live' **DDSDataWriter** (p. 1113) entities writing the instance. That is, all existing writers either unregister the instance (call unregister) or lose their liveliness,

and

3. A time interval longer that **DDS_DurabilityServiceQosPolicy::service_cleanup_delay** (p. 619) has elapsed since the moment RTI Connext detected that the previous two conditions were met.

The utility of **DDS_DurabilityServiceQosPolicy::service_cleanup_delay** (p. 619) is apparent in the situation where an application disposes an instance and it crashes before it has a chance to complete additional tasks related to the disposition. Upon restart, the application may ask for initial data to regain its state and the delay introduced by the service_cleanup_delay will allow the restarted application to receive the information on the disposed instance and complete the interrupted tasks.

6.34.3 Compatibility

The value offered is considered compatible with the value requested if and only if the inequality *offered kind* >= *requested kind* evaluates to 'TRUE'.

For the purposes of this inequality, the values of DURABILITY kind are considered ordered such that **DDS_VOLATILE_DURABILITY_QOS** (p. 349) < **DDS_TRANSIENT_LOCAL_DURABILITY_QOS** (p. 349) < **DDS_TRANSIENT_DURABILITY_QOS** (p. 349) < **DDS_PERSISTENT_DURABILITY_QOS** (p. 349).

6.34.4 Member Data Documentation

6.34.4.1 DDS_DurabilityQosPolicyKind DDS_DurabilityQosPolicy::kind

The kind of durability.

[**default**] **DDS_VOLATILE_DURABILITY_QOS** (p. 349)

6.34.4.2 DDS_Boolean DDS_DurabilityQosPolicy::direct_communication

*<<**eXtension**>>* *(p. 199)* Indicates whether or not a TRANSIENT or PERSISTENT **DDSDataReader** (p. 1087) should receive samples directly from a TRANSIENT or PERSISTENT **DDSDataWriter** (p. 1113)

When direct_communication is set to **DDS_BOOLEAN_TRUE** (p. 298), a TRANSIENT or PERSISTENT **DDSDataReader** (p. 1087) will receive samples from both the original **DDSDataWriter** (p. 1113) configured with TRANSIENT or PERSISTENT durability and the **DDSDataWriter** (p. 1113) created by the persistence service. This peer-to-peer communication pattern provides low latency between end-points.

If the same sample is received from the original **DDSDataWriter** (p. 1113) and the persistence service, the middleware will discard the duplicate.

When direct_communication is set to **DDS_BOOLEAN_FALSE** (p. 299), a TRANSIENT or PERSISTENT **DDSDataReader** (p. 1087) will only receive samples from the **DDSDataWriter** (p. 1113) created by the persistence service. This brokered communication pattern provides a way to guarantee eventual consistency.

[**default**] **DDS_BOOLEAN_TRUE** (p. 298)

6.35 DDS_DurabilityServiceQosPolicy Struct Reference

Various settings to configure the external *RTI Persistence Service* used by RTI Connext for DataWriters with a **DDS_DurabilityQosPolicy** (p. 614) setting of **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) or **DDS_TRANSIENT_DURABILITY_QOS** (p. 349).

Public Attributes

- struct **DDS_Duration_t service_cleanup_delay**

  *[**Not supported (optional)**] Controls when the service is able to remove all information regarding a data instances.*

- **DDS_HistoryQosPolicyKind history_kind**

  The kind of history to apply in recouping durable data.

- **DDS_Long history_depth**

  Part of history QoS policy to apply when feeding a late joiner.

- **DDS_Long max_samples**

  Part of resource limits QoS policy to apply when feeding a late joiner.

- **DDS_Long max_instances**

  Part of resource limits QoS policy to apply when feeding a late joiner.

- **DDS_Long max_samples_per_instance**

  Part of resource limits QoS policy to apply when feeding a late joiner.

6.35.1 Detailed Description

Various settings to configure the external *RTI Persistence Service* used by RTI Connext for DataWriters with a **DDS_DurabilityQosPolicy** (p. 614) setting of **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) or **DDS_TRANSIENT_DURABILITY_QOS** (p. 349).

Entity:

DDSTopic (p. 1419), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

See also:

DURABILITY (p. 348)
HISTORY (p. 367)
RESOURCE_LIMITS (p. 371)

6.35.2 Usage

When a DataWriter's **DDS_DurabilityQosPolicy::kind** (p. 617) is **DDS_PERSISTENT_DURABILITY_QOS** (p. 349) or **DDS_TRANSIENT_DURABILITY_QOS** (p. 349), an external service, the *RTI Persistence Service*, is used to store and possibly forward the data sent by the **DDSDataWriter** (p. 1113) to **DDSDataReader** (p. 1087) objects that are created *after* the data was initially sent.

This QoS policy is used to configure certain parameters of the Persistence Service when it operates on the behalf of the **DDSDataWriter** (p. 1113), such as how much data to store. For example, it configures the **HISTORY** (p. 367) and the **RESOURCE_LIMITS** (p. 371) used by the fictitious DataReader and DataWriter used by the Persistence Service. Note, however, that the Persistence Service itself may be configured to ignore these values and instead use values from its own configuration file.

6.35.3 Member Data Documentation

6.35.3.1 struct DDS_Duration_t DDS_DurabilityServiceQosPolicy::service_cleanup_delay [read]

[**Not supported (optional)**] Controls when the service is able to remove all information regarding a data instances.

[**default**] 0

6.35.3.2 DDS_HistoryQosPolicyKind DDS_DurabilityServiceQosPolicy::history_kind

The kind of history to apply in recouping durable data.

[**default**] **DDS_KEEP_LAST_HISTORY_QOS** (p. 368)

6.35.3.3 DDS_Long DDS_DurabilityServiceQosPolicy::history_depth

Part of history QoS policy to apply when feeding a late joiner.

[default] 1

6.35.3.4 DDS_Long DDS_DurabilityServiceQosPolicy::max_samples

Part of resource limits QoS policy to apply when feeding a late joiner.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

6.35.3.5 DDS_Long DDS_DurabilityServiceQosPolicy::max_instances

Part of resource limits QoS policy to apply when feeding a late joiner.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

6.35.3.6 DDS_Long DDS_DurabilityServiceQosPolicy::max_samples_per_instance

Part of resource limits QoS policy to apply when feeding a late joiner.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

6.36 DDS_Duration_t Struct Reference

Type for *duration* representation.

Public Attributes

- **DDS_Long sec**
  seconds
- **DDS_UnsignedLong nanosec**
  nanoseconds

6.36.1 Detailed Description

Type for *duration* representation.

Represents a time interval.

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

6.36.2 Member Data Documentation

6.36.2.1 DDS_Long DDS_Duration_t::sec

seconds

Examples:

HelloWorld_subscriber.cxx.

6.36.2.2 DDS_UnsignedLong DDS_Duration_t::nanosec

nanoseconds

6.37 DDS_DynamicData Struct Reference

A sample of any complex data type, which can be inspected and manipulated reflectively.

Public Member Functions

- **DDS_DynamicData** (const **DDS_TypeCode** *type, const **DDS_DynamicDataProperty_t** &property)

  *The constructor for new **DDS_DynamicData** (p. 622) objects.*

- **DDS_Boolean is_valid** () const

  Indicates whether the object was constructed properly.

- **DDS_ReturnCode_t copy** (const **DDS_DynamicData** &src)

  Deeply copy from the given object to this object.

- **DDS_Boolean equal** (const **DDS_DynamicData** &other) const

  *Indicate whether the contents of another **DDS_DynamicData** (p. 622) sample are the same as those of this one.*

- **DDS_DynamicData** & **operator=** (const **DDS_DynamicData** &src)

  Deeply copy from the given object to this object.

- **DDS_Boolean operator==** (const **DDS_DynamicData** &other) const

  *Indicate whether the contents of another **DDS_DynamicData** (p. 622) sample are the same as those of this one.*

- **DDS_ReturnCode_t clear_all_members** ()

  Clear the contents of all data members of this object, including key members.

- **DDS_ReturnCode_t clear_nonkey_members** ()

  Clear the contents of all data members of this object, not including key members.

- **DDS_ReturnCode_t clear_member** (const char *member_name, **DDS_DynamicDataMemberId** member_id)

  Clear the contents of a single data member of this object.

- **DDS_ReturnCode_t set_buffer** (**DDS_Octet** *storage, **DDS_Long** size)

Associate a buffer with this dynamic data object.

- **DDS_ReturnCode_t get_estimated_max_buffer_size** (**DDS_Long** &size)

  Get the estimated maximum buffer size for a DynamicData object.

- **DDS_ReturnCode_t print** (FILE *fp, int indent) const

  Output a textual representation of this object and its contents to the given file.

- void **get_info** (**DDS_DynamicDataInfo** &info_out) const

  *Fill in the given descriptor with information about this **DDS_DynamicData** (p. 622).*

- **DDS_ReturnCode_t bind_type** (const **DDS_TypeCode** *type)

  *If this **DDS_DynamicData** (p. 622) object is not yet associated with a data type, set that type now to the given **DDS_TypeCode** (p. 992).*

- **DDS_ReturnCode_t unbind_type** ()

  *Dissociate this **DDS_DynamicData** (p. 622) object from any particular data type.*

- **DDS_ReturnCode_t bind_complex_member** (**DDS_DynamicData** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id)

  *Use another **DDS_DynamicData** (p. 622) object to provide access to a complex field of this **DDS_DynamicData** (p. 622) object.*

- **DDS_ReturnCode_t unbind_complex_member** (**DDS_DynamicData** &value)

  *Tear down the association created by a **DDS_DynamicData::bind_complex_member** (p. 647) operation, committing any changes to the outer object since then.*

- const **DDS_TypeCode** * **get_type** () const

  *Get the data type, of which this **DDS_DynamicData** (p. 622) represents an instance.*

- **DDS_TCKind get_type_kind** () const

  Get the kind of this object's data type.

- **DDS_UnsignedLong get_member_count** () const

  Get the number of members in this sample.

- **DDS_Boolean member_exists** (const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Indicates whether a member of a particular name/ID exists in this data sample.

- **DDS_Boolean member_exists_in_type** (const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Indicates whether a member of a particular name/ID exists in this data sample's type.

- **DDS_ReturnCode_t get_member_info** (**DDS_DynamicDataMemberInfo** &info, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Fill in the given descriptor with information about the identified member of this **DDS_DynamicData** (p. 622) sample.*

- **DDS_ReturnCode_t get_member_info_by_index** (struct **DDS_DynamicDataMemberInfo** &info, **DDS_UnsignedLong** index) const

  *Fill in the given descriptor with information about the identified member of this **DDS_DynamicData** (p. 622) sample.*

- **DDS_ReturnCode_t get_member_type** (const **DDS_TypeCode** *&type_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get the type of the given member of this sample.

- **DDS_ReturnCode_t is_member_key** (**DDS_Boolean** &is_key_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Indicates whether a given member forms part of the key of this sample's data type.

- **DDS_ReturnCode_t get_long** (**DDS_Long** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Long** (p. 300) or another type implicitly convertible to it (**DDS_Octet** (p. 299), **DDS_Char** (p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), or **DDS_Enum** (p. 301)).*

- **DDS_ReturnCode_t get_short** (**DDS_Short** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Short** (p. 299) or another type implicitly convertible to it (**DDS_Octet** (p. 299) or **DDS_Char** (p. 299)).*

- **DDS_ReturnCode_t get_ulong** (**DDS_UnsignedLong** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_UnsignedLong*** (p. 300) *or another type implicitly convertible to it (**DDS_Octet*** (p. 299), ***DDS_Char*** (p. 299), ***DDS_Short*** (p. 299), ***DDS_UnsignedShort*** (p. 299), *or **DDS_Enum*** (p. 301)).

- **DDS_ReturnCode_t get_ushort** (**DDS_UnsignedShort** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_UnsignedShort*** (p. 299) *or another type implicitly convertible to it (**DDS_Octet*** (p. 299) *or **DDS_Char*** (p. 299)).

- **DDS_ReturnCode_t get_float** (**DDS_Float** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Float*** (p. 300).

- **DDS_ReturnCode_t get_double** (**DDS_Double** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Double*** (p. 300) *or another type implicitly convertible to it (**DDS_Float*** (p. 300)).

- **DDS_ReturnCode_t get_boolean** (**DDS_Boolean** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Boolean*** (p. 301).

- **DDS_ReturnCode_t get_char** (**DDS_Char** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Char*** (p. 299).

- **DDS_ReturnCode_t get_octet** (**DDS_Octet** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Octet*** (p. 299).

- **DDS_ReturnCode_t get_longlong** (**DDS_LongLong** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_LongLong*** (p. 300) *or another type implicitly convertible to it (**DDS_Octet*** (p. 299), ***DDS_Char***

*(p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), **DDS_Long** (p. 300), **DDS_UnsignedLong** (p. 300), or **DDS_Enum** (p. 301)).*

- **DDS_ReturnCode_t get_ulonglong** (**DDS_UnsignedLongLong** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_UnsignedLongLong** (p. 300) or another type implicitly convertible to it (**DDS_Octet** (p. 299), **DDS_Char** (p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), **DDS_Long** (p. 300), **DDS_UnsignedLong** (p. 300), or **DDS_Enum** (p. 301)).*

- **DDS_ReturnCode_t get_longdouble** (**DDS_LongDouble** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_LongDouble** (p. 300) or another type implicitly convertible to it (**DDS_Float** (p. 300) or **DDS_Double** (p. 300)).*

- **DDS_ReturnCode_t get_wchar** (**DDS_Wchar** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Wchar** (p. 299) or another type implicitly convertible to it (**DDS_Char** (p. 299)).*

- **DDS_ReturnCode_t get_string** (char *&value, **DDS_UnsignedLong** *size, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get the value of the given field, which is of type char.*

- **DDS_ReturnCode_t get_wstring** (**DDS_Wchar** *&value, **DDS_UnsignedLong** *size, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  *Get the value of the given field, which is of type **DDS_Wchar** (p. 299)*.*

- **DDS_ReturnCode_t get_complex_member** (**DDS_DynamicData** &value_out, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the value of the given field, which is of some composed type.

- **DDS_ReturnCode_t get_long_array** (**DDS_Long** *array, **DDS_UnsignedLong** *length, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_short_array** (**DDS_Short** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_ulong_array** (**DDS_UnsignedLong** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_ushort_array** (**DDS_UnsignedShort** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_float_array** (**DDS_Float** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_double_array** (**DDS_Double** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_boolean_array** (**DDS_Boolean** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_char_array** (**DDS_Char** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_octet_array** (**DDS_Octet** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

ˆ **DDS_ReturnCode_t get_longlong_array** (**DDS_LongLong** ∗array, **DDS_UnsignedLong** ∗length, const char ∗member_name, **DDS_DynamicDataMemberId** member_id) const

Get a copy of the given array member.

- **DDS_ReturnCode_t get_ulonglong_array** (**DDS_UnsignedLongLong** *array, **DDS_UnsignedLong** *length, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given array member.

- **DDS_ReturnCode_t get_longdouble_array** (**DDS_LongDouble** *array, **DDS_UnsignedLong** *length, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given array member.

- **DDS_ReturnCode_t get_wchar_array** (**DDS_Wchar** *array, **DDS_UnsignedLong** *length, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given array member.

- **DDS_ReturnCode_t get_long_seq** (**DDS_LongSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_short_seq** (**DDS_ShortSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_ulong_seq** (**DDS_UnsignedLongSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_ushort_seq** (**DDS_UnsignedShortSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_float_seq** (**DDS_FloatSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_double_seq** (**DDS_DoubleSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_boolean_seq** (**DDS_BooleanSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_char_seq** (**DDS_CharSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_octet_seq** (**DDS_OctetSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_longlong_seq** (**DDS_LongLongSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_ulonglong_seq** (**DDS_UnsignedLongLongSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_longdouble_seq** (**DDS_LongDoubleSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t get_wchar_seq** (**DDS_WcharSeq** &seq, const char *member_name, **DDS_DynamicDataMemberId** member_id) const

  Get a copy of the given sequence member.

- **DDS_ReturnCode_t set_long** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Long** value)

  *Set the value of the given field, which is of type **DDS_Long** (p. 300).*

- **DDS_ReturnCode_t set_short** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Short** value)

  *Set the value of the given field, which is of type **DDS_Short** (p. 299).*

- **DDS_ReturnCode_t set_ulong** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** value)

*Set the value of the given field, which is of type **DDS_UnsignedLong** (p. 300).*

- **DDS_ReturnCode_t set_ushort** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedShort** value)

  *Set the value of the given field, which is of type **DDS_UnsignedShort** (p. 299).*

- **DDS_ReturnCode_t set_float** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Float** value)

  *Set the value of the given field, which is of type **DDS_Float** (p. 300).*

- **DDS_ReturnCode_t set_double** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Double** value)

  *Set the value of the given field, which is of type **DDS_Double** (p. 300).*

- **DDS_ReturnCode_t set_boolean** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Boolean** value)

  *Set the value of the given field, which is of type **DDS_Boolean** (p. 301).*

- **DDS_ReturnCode_t set_char** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Char** value)

  *Set the value of the given field, which is of type **DDS_Char** (p. 299).*

- **DDS_ReturnCode_t set_octet** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Octet** value)

  *Set the value of the given field, which is of type **DDS_Octet** (p. 299).*

- **DDS_ReturnCode_t set_longlong** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_LongLong** value)

  *Set the value of the given field, which is of type **DDS_LongLong** (p. 300).*

- **DDS_ReturnCode_t set_ulonglong** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLongLong** value)

  *Set the value of the given field, which is of type **DDS_UnsignedLongLong** (p. 300).*

- **DDS_ReturnCode_t set_longdouble** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_LongDouble** value)

  *Set the value of the given field, which is of type **DDS_LongDouble** (p. 300).*

ˆ **DDS_ReturnCode_t set_wchar** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_Wchar** value)

*Set the value of the given field, which is of type **DDS_Wchar** (p. 299).*

ˆ **DDS_ReturnCode_t set_string** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const char *value)

Set the value of the given field of type char.*

ˆ **DDS_ReturnCode_t set_wstring** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_Wchar** *value)

*Set the value of the given field of type **DDS_Wchar** (p. 299)*.*

ˆ **DDS_ReturnCode_t set_complex_member** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_DynamicData** &value)

*Copy the state of the given **DDS_DynamicData** (p. 622) object into a member of this object.*

ˆ **DDS_ReturnCode_t set_long_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Long** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_short_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Short** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_ulong_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_UnsignedLong** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_ushort_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_UnsignedShort** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_float_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Float** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_double_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Double** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_boolean_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Boolean** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_char_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Char** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_octet_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Octet** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_longlong_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_LongLong** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_ulonglong_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_UnsignedLongLong** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_longdouble_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_LongDouble** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_wchar_array** (const char *member_name, **DDS_DynamicDataMemberId** member_id, **DDS_UnsignedLong** length, const **DDS_Wchar** *array)

Set the contents of the given array member.

ˆ **DDS_ReturnCode_t set_long_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_LongSeq** &value)

Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_short_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_ShortSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_ulong_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_UnsignedLongSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_ushort_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_UnsignedShortSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_float_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_FloatSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_double_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_DoubleSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_boolean_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_BooleanSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_char_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_CharSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_octet_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_OctetSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_longlong_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_LongLongSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_ulonglong_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_UnsignedLongLongSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_longdouble_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_LongDoubleSeq** &value)

  Set the contents of the given sequence member.

- **DDS_ReturnCode_t set_wchar_seq** (const char *member_name, **DDS_DynamicDataMemberId** member_id, const **DDS_WcharSeq** &value)

  Set the contents of the given sequence member.

6.37.1 Detailed Description

A sample of any complex data type, which can be inspected and manipulated reflectively.

Objects of type **DDS_DynamicData** (p. 622) represent corresponding objects of the type identified by their **DDS_TypeCode** (p. 992). Because the definition of these types may not have existed at compile time on the system on which the application is running, you will interact with the data using an API of reflective getters and setters.

For example, if you had access to your data types at compile time, you could do this:

```
theValue = theObject.theField;
```

Instead, you will do something like this:

```
theValue = get(theObject, "theField");
```

DDS_DynamicData (p. 622) objects can represent any complex data type, including those of type kinds **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_VALUE** (p. 67), and **DDS_TK_SPARSE** (p. 67). They cannot represent objects of basic types (e.g. integers and strings). Since those type definitions always exist on every system, you can examine their objects directly.

6.37.2 Member Names and IDs

The members of a data type can be identified in one of two ways: by their name or by their numeric ID. The former is often more transparent to human users; the latter is typically faster.

You define the name and ID of a type member when you add that member to that type. When you define a sparse type, you will typically choose both explicitly. If you define your type in IDL or XML, the name will be the field name that appears in the type definition; the ID will be the one-based index of the field in declaration order. For example, in the following IDL structure, the ID of `theLong` is 2.

```
struct MyType {
     short theShort;
     long theLong;
 };
```

For unions (**DDS_TK_UNION** (p. 66)), the ID of a member is the discriminator value corresponding to that member.

6.37.3 Arrays and Sequences

The "members" of array and sequence types, unlike those of structure and union types, don't have names or explicit member IDs. However, they may nevertheless be accessed by "ID": the ID is one more than the index. (The first element has ID 1, the second 2, etc.)

Multi-dimensional arrays are effectively flattened by this pattern. For example, for an array `theArray`[4][5], accessing ID 7 is equivalent to index 6, or the second element of the second group of 5.

To determine the length of a collection-typed member of a structure or union, you have two choices:

1. Get the length along with the data: call the appropriate array accessor (see **Getters and Setters** (p. 637)) and check the resulting length.
2. Get the length without getting the data itself: call **DDS_DynamicData::get_member_info** (p. 652) and check the resulting **DDS_DynamicDataMemberInfo::element_count** (p. 723).

6.37.4 Available Functionality

The Dynamic Data API is large when measured by the number of methods it contains. But each method falls into one of a very small number of categories.

You will find it easier to navigate this documentation if you understand these categories.

6.37.4.1 Lifecycle and Utility Methods

Managing the lifecycle of **DDS_DynamicData** (p. 622) objects is simple. You have two choices:

1. Usually, you will go through a **DDSDynamicDataTypeSupport** (p. 1246) factory object, which will ensure that the type and property information for the new **DDS_DynamicData** (p. 622) object corresponds to a registered type in your system.
2. In certain advanced cases, such as when you're navigating a nested structure, you will want to have a **DDS_DynamicData** (p. 622) object that is not bound up front to any particular type, or you will want to initialize the object in a custom way. In that case, you can call the constructor directly.

DDSDynamicDataTypeSupport (p. 1246)	**DDS_DynamicData** (p. 622)
DDSDynamicDataTypeSupport::create_data (p. 1250)	DDS_DynamicData::DDS_DynamicData
DDSDynamicDataTypeSupport::delete_data (p. 1251)	DDS_DynamicData::~DDS_DynamicData

Table 6.1: Lifecycle

You can also copy **DDS_DynamicData** (p. 622) objects:

- **DDS_DynamicData::copy** (p. 641)
- **DDS_DynamicData::operator=** (p. 642)

You can test them for equality:

- **DDS_DynamicData::equal** (p. 641)
- **DDS_DynamicData::operator==** (p. 642)

And you can print their contents:

- **DDS_DynamicData::print** (p. 645)
- **DDSDynamicDataTypeSupport::print_data** (p. 1251)

6.37.4.2 Getters and Setters

Most methods get or set the value of some field. These methods are named according to the type of the field they access.

The names of integer types vary across languages. The programming API for each language reflects that programming language. However, if your chosen language does not use the same names as the language that you used to define your types (e.g., IDL), or if you need to interoperate among programming languages, you will need to understand these differences. They are explained the following table.

Type	IDL	C, C++	C#	Java
16-bit integer	short	DDS_Short	short	short
32-bit integer	long	DDS_Long	int	int
64-bit integer	long long	DDS_-LongLong	long	long

Table 6.2: Integer Type Names Across Languages

When working with a **DDS_DynamicData** (p. 622) object representing an array or sequence, calling one of the "get" methods below for an index that is out of bounds will result in **DDS_RETCODE_NO_DATA** (p. 316). Calling "set" for an index that is past the end of a sequence will cause that sequence to automatically lengthen (filling with default contents).

In addition to getting or setting a field, you can "clear" its value; that is, set it to a default zero value.

- **DDS_DynamicData::clear_member** (p. 643)
- **DDS_DynamicData::clear_all_members** (p. 642)
- **DDS_DynamicData::clear_nonkey_members** (p. 643)

6.37.4.3 Query and Iteration

Not all components of your application will have static knowledge of all of the fields of your type. Sometimes, you will want to query meta-data about the fields that appear in a given data sample.

- **DDS_DynamicData::get_type** (p. 650)
- **DDS_DynamicData::get_type_kind** (p. 650)
- **DDS_DynamicData::get_member_type** (p. 654)

- **DDS_DynamicData::get_member_info** (p. 652)
- **DDS_DynamicData::get_member_count** (p. 650)
- **DDS_DynamicData::get_member_info_by_index** (p. 653)
- **DDS_DynamicData::member_exists** (p. 651)
- **DDS_DynamicData::member_exists_in_type** (p. 652)
- **DDS_DynamicData::is_member_key** (p. 654)

6.37.4.4 Type/Object Association

Sometimes, you may want to change the association between a data object and its type. This is not something you can do with a typical object, but with **DDS_DynamicData** (p. 622) objects, it is a powerful capability. It allows you to, for example, examine nested structures without copying them by using a "bound" **DDS_DynamicData** (p. 622) object as a view into an enclosing **DDS_DynamicData** (p. 622) object.

- **DDS_DynamicData::bind_type** (p. 646)
- **DDS_DynamicData::unbind_type** (p. 647)
- **DDS_DynamicData::bind_complex_member** (p. 647)
- **DDS_DynamicData::unbind_complex_member** (p. 649)

6.37.4.5 Keys

Keys can be specified in dynamically defined types just as they can in types defined in generated code. However, there are some minor restrictions when *sparse value types* are involved (see **DDS_TK_SPARSE** (p. 67)).

- If a type has a member that is of a sparse value type, that member cannot be a key for the enclosing type.
- Sparse value types themselves may have at most a single key field. That field may itself be of any type.

6.37.5 Performance

Due to the way in which **DDS_DynamicData** (p. 622) objects manage their internal state, it is typically more efficient, when setting the field values of a **DDS_DynamicData** (p. 622) for the first time, to do so in the declared order of those fields.

For example, suppose a type definition like the following:

```
struct MyType {
     float my_float;
     sequence<octet> my_bytes;
     short my_short;
 };
```

The richness of the type system makes it difficult to fully characterize the performance differences between all access patterns. Nevertheless, the following are generally true:

- It will be most performant to set the value of `my_float`, then `my_bytes`, and finally `my_short`.
- The order of modification has a greater impact for types of kind **DDS_TK_STRUCT** (p. 66) and **DDS_TK_VALUE** (p. 67) than it does for types of kind **DDS_TK_SPARSE** (p. 67).
- Modifications to variable-sized types (i.e. those containing strings, sequences, unions, or optional members) are more expensive than modifications to fixed-size types.

MT Safety:

UNSAFE. In general, using a single **DDS_DynamicData** (p. 622) object concurrently from multiple threads is *unsafe*.

6.37.6 Constructor & Destructor Documentation

6.37.6.1 DDS_DynamicData::DDS_DynamicData (const DDS_TypeCode * *type*, const DDS_DynamicDataProperty_t & *property*)

The constructor for new **DDS_DynamicData** (p. 622) objects.

The type parameter may be NULL. In that case, this **DDS_DynamicData** (p. 622) must be *bound* with **DDS_DynamicData::bind_type** (p. 646) or **DDS_DynamicData::bind_complex_member** (p. 647) before it can be used.

If the **DDS_TypeCode** (p. 992) is not NULL, the newly constructed **DDS_DynamicData** (p. 622) object will retain a reference to it. It is *not* safe to delete the **DDS_TypeCode** (p. 992) until all samples that use it have themselves been deleted.

In most cases, it is not necessary to call this constructor explicitly. Instead, use **DDSDynamicDataTypeSupport::create_data** (p. 1250), and the **DDS_TypeCode** (p. 992) and properties will be specified for you. Using the factory

method also ensures that the memory management contract documented above is followed correctly, because the **DDSDynamicDataTypeSupport** (p. 1246) object maintains the **DDS_TypeCode** (p. 992) used by the samples it creates.

However you create a **DDS_DynamicData** (p. 622) object, you must delete it when you are finished with it. If you choose to use this constructor, delete the object with the destructor: DDS_DynamicData::~DDS_DynamicData.

```
DDS_DynamicData* sample = new DDS_DynamicData(
     myType, myProperties);
 // Failure indicated by is_valid() method.
 // Do something...
 delete sample;
```

NOTE that RTI Connext does not explicitly generate any exceptions in this constructor, because C++ exception support is not consistent across all platforms on which RTI Connext runs. Therefore, to check whether construction succeeded, you must use the **DDS_DynamicData::is_valid** (p. 640) method. Alternatively, you can create an **DDS_DynamicData** (p. 622) object with **DDSDynamicDataTypeSupport::create_data** (p. 1250), which returns NULL on failure, eliminating the need to call **DDS_DynamicData::is_valid** (p. 640).

Parameters:

type *<<in>> (p. 200)* The type of which the new object will represent an object.

property *<<in>> (p. 200)* Properties that configure the behavior of the new object. Most users can simply use **DDS_DYNAMIC_DATA_PROPERTY_DEFAULT** (p. 80).

See also:

DDS_DynamicData::is_valid (p. 640)
DDS_DynamicData::~DDS_DynamicData
DDSDynamicDataTypeSupport::create_data (p. 1250)

6.37.7 Member Function Documentation

6.37.7.1 DDS_Boolean DDS_DynamicData::is_valid () const

Indicates whether the object was constructed properly.

This method returns **DDS_BOOLEAN_TRUE** (p. 298) if the constructor succeeded; it returns **DDS_BOOLEAN_FALSE** (p. 299) if the constructor failed for any reason, which should also have resulted in a log message. It is only necessary to call this method if you created the **DDS_DynamicData** (p. 622) object using the constructor, DDS_DynamicData::DDS_DynamicData.

Possible failure reasons include passing an invalid type or invalid properties to the constructor.

This method is necessary because C++ exception support is not consistent across all of the platforms on which RTI Connext runs. Therefore, the implementation does not throw any exceptions in the constructor.

MT Safety:

UNSAFE.

See also:

DDS_DynamicData::DDS_DynamicData

6.37.7.2 DDS_ReturnCode_t DDS_DynamicData::copy (const DDS_DynamicData & *src*)

Deeply copy from the given object to this object.

MT Safety:

UNSAFE.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::operator= (p. 642)

6.37.7.3 DDS_Boolean DDS_DynamicData::equal (const DDS_DynamicData & *other*) const

Indicate whether the contents of another **DDS_DynamicData** (p. 622) sample are the same as those of this one.

This operation compares the data and type of existing members. The types of non-instantiated members may differ in sparse types.

MT Safety:

UNSAFE.

See also:

DDS_DynamicData::operator== (p. 642)
DDS_TK_SPARSE (p. 67)

6.37.7.4 DDS_DynamicData& DDS_DynamicData::operator= (const DDS_DynamicData & *src*)

Deeply copy from the given object to this object.

MT Safety:

UNSAFE.

See also:

DDS_DynamicData::copy (p. 641)

6.37.7.5 DDS_Boolean DDS_DynamicData::operator== (const DDS_DynamicData & *other*) const

Indicate whether the contents of another **DDS_DynamicData** (p. 622) sample are the same as those of this one.

This operation compares the data and type of existing members. The types of non-instantiated members may differ in sparse types.

MT Safety:

UNSAFE.

See also:

DDS_DynamicData::equal (p. 641)
DDS_TK_SPARSE (p. 67)

6.37.7.6 DDS_ReturnCode_t DDS_DynamicData::clear_all_members ()

Clear the contents of all data members of this object, including key members.

MT Safety:

UNSAFE.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::clear_nonkey_members (p. 643)
DDS_DynamicData::clear_member (p. 643)

6.37.7.7 DDS_ReturnCode_t DDS_DynamicData::clear_nonkey_members ()

Clear the contents of all data members of this object, not including key members.

This method is only applicable to sparse value types.

MT Safety:

UNSAFE.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_TK_SPARSE (p. 67)
DDS_DynamicData::clear_all_members (p. 642)
DDS_DynamicData::clear_member (p. 643)

6.37.7.8 DDS_ReturnCode_t DDS_DynamicData::clear_member (const char * *member_name*, DDS_DynamicDataMemberId *member_id*)

Clear the contents of a single data member of this object.

This method is only applicable to sparse value types.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_TK_SPARSE (p. 67)
DDS_DynamicData::clear_all_members (p. 642)
DDS_DynamicData::clear_nonkey_members (p. 643)

6.37.7.9 DDS_ReturnCode_t DDS_DynamicData::set_buffer (DDS_Octet * *storage*, DDS_Long *size*)

Associate a buffer with this dynamic data object.

The DynamicData object will use the input buffer to store its value.

If the DynamicData object already has a value, this function will clear the contents of all data members.

The memory of the buffer being replaced will be freed, unless the buffer was assigned by a previous call to this method.

If the size of the input buffer is not big enough to hold the future value of the DynamicData object, the set operations will return **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) when the size is exceeded.

The internal representation of a DynamicData object is based on CDR encapsulation and introduces additional overhead beyond the data representation of the equivalent C structure. When providing an input buffer, try to account for that overhead.

The method **DDS_DynamicData::get_estimated_max_buffer_size** (p. 645) can be used to obtain an estimated maximum size for the input buffer. This size is calculated based on an assignment pattern where the value of a data member is set a single time between calls to **DDS_DynamicData::clear_all_members** (p. 642).

MT Safety:

UNSAFE.

Parameters:

storage *<<in>>* *(p. 200)* Storage buffer.

size *<<**in**>>* *(p. 200)* Buffer size in bytes.

Returns:

One of the **Standard Return Codes** (p. 314)

6.37.7.10 DDS_ReturnCode_t DDS_DynamicData::get_estimated_max_buffer_size (DDS_Long & *size*)

Get the estimated maximum buffer size for a DynamicData object.

This method gets the estimated maximum buffer size of a DynamicData object based on the associated type.

The output size is calculated based on an assignment pattern where the value of a data member is set a single time between calls to **DDS_DynamicData::clear_all_members** (p. 642).

If this is not the case (for example, the value of a string member is set multiple times), the maximum size returned by this method may not be representative.

This method can be used to obtain the size of the buffer provided to the method **DDS_DynamicData::set_buffer** (p. 644).

MT Safety:

UNSAFE.

Parameters:

size *<<**out**>>* *(p. 200)* Estimated maximum buffer size in bytes.

Returns:

One of the **Standard Return Codes** (p. 314)

6.37.7.11 DDS_ReturnCode_t DDS_DynamicData::print (FILE * *fp*, int *indent*) const

Output a textual representation of this object and its contents to the given file.

This method is equivalent to **DDSDynamicDataTypeSupport::print_data** (p. 1251).

Warning: This operation may not display any data for members at the end of a data structure that have not been explicitly set before the data sample is serialized. This will not be a problem on a received data sample, which should always correctly display all members.

MT Safety:

UNSAFE.

Parameters:

fp *<<**in**>>* *(p. 200)* The file to which the object should be printed.

indent *<<**in**>>* *(p. 200)* The output of this method will be pretty-printed. This argument indicates the amount of initial indentation of the output.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDynamicDataTypeSupport::print_data (p. 1251)

6.37.7.12 void DDS_DynamicData::get_info (DDS_DynamicDataInfo & *info_out*) const

Fill in the given descriptor with information about this **DDS_DynamicData** (p. 622).

MT Safety:

UNSAFE.

Parameters:

info_out *<<**out**>>* *(p. 200)* The descriptor object whose contents will be overwritten by this operation.

6.37.7.13 DDS_ReturnCode_t DDS_DynamicData::bind_type (const DDS_TypeCode * *type*)

If this **DDS_DynamicData** (p. 622) object is not yet associated with a data type, set that type now to the given **DDS_TypeCode** (p. 992).

This advanced operation allows you to reuse a single **DDS_DynamicData** (p. 622) object with multiple data types.

```
DDS_DynamicData* myData = new DDS_DynamicData(NULL, myProperties);
 DDS_TypeCode* myType = ...;
 myData->bind_type(myType);
 // Do something...
 myData->unbind_type();
 delete myData;
```

Note that the **DDS_DynamicData** (p. 622) object will retain a reference to the **DDS_TypeCode** (p. 992) object you provide. It is *not* safe to delete the **DDS_TypeCode** (p. 992) until after it is unbound.

MT Safety:

UNSAFE.

Parameters:

type <<***in***>> *(p. 200)* The type to associate with this **DDS_DynamicData** (p. 622) object.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::unbind_type (p. 647)

6.37.7.14 DDS_ReturnCode_t DDS_DynamicData::unbind_type ()

Dissociate this **DDS_DynamicData** (p. 622) object from any particular data type.

This step is necessary before the object can be associated with a new data type.

This operation clears all members as a side effect.

MT Safety:

UNSAFE.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::bind_type (p. 646)
DDS_DynamicData::clear_all_members (p. 642)

6.37.7.15 DDS_ReturnCode_t DDS_DynamicData::bind_complex_member (DDS_DynamicData & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*)

Use another **DDS_DynamicData** (p. 622) object to provide access to a complex field of this **DDS_DynamicData** (p. 622) object.

For example, consider the following data types:

```
struct MyFieldType {
     float theFloat;
 };

 struct MyOuterType {
     MyFieldType complexMember;
 };
```

Suppose you have an instance of `MyOuterType`, and you would like to examine the contents of its member `complexMember`. To do this, you must *bind* another **DDS_DynamicData** (p. 622) object to that member. This operation will bind the type code of the member to the provided **DDS_DynamicData** (p. 622) object and perform additional initialization.

The following example demonstrates the usage pattern. Note that error handling has been omitted for brevity.

```
DDS_DynamicData outer = ...;
 DDS_DynamicData toBeBound(NULL, myProperties);
 outer.bind_complex_member(
     toBeBound,
     "complexMember",
     DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
 float theFloatValue = 0.0f;
 toBeBound.get_float(
     theFloat,
     "theFloat"
     DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
 outer.unbind_complex_member(toBeBound);
```

This operation is only permitted when the object `toBeBound` (named as in the example above) is not currently associated with any type, including already being bound to another member. You can see in the example that this object is created directly with the constructer and is not provided with a **DDS_TypeCode** (p. 992).

Only a single member of a given **DDS_DynamicData** (p. 622) object may be bound at one time – however, members *of* members may be recursively bound to any depth. Furthermore, while the outer object has a bound member, it may only be modified through that bound member. That is, after calling this member, all "set" operations on the outer object will be disabled until **DDS_DynamicData::unbind_complex_member** (p. 649) has been called. Furthermore, any bound member must be unbound before a sample can be written or deleted.

This method is logically related to **DDS_DynamicData::get_complex_member** (p. 666) in that both allow you to examine the state of nested objects. They are different in an important way: this method provides a view into an

outer object, such that any change made to the inner object will be reflected in the outer. But the **DDS_DynamicData::get_complex_member** (p. 666) operation *copies* the state of the nested object; changes to it will not be reflected in the source object.

Note that you can bind to a member of a sequence at an index that is past the current length of that sequence. In that case, this method behaves like a "set" method: it automatically lengthens the sequence (filling in default elements) to allow the bind to take place. See **Getters and Setters** (p. 637).

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* The object that you wish to bind to the field.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::unbind_complex_member (p. 649)
DDS_DynamicData::get_complex_member (p. 666)

6.37.7.16 DDS_ReturnCode_t DDS_DynamicData::unbind_complex_member (DDS_DynamicData & *value*)

Tear down the association created by a **DDS_DynamicData::bind_complex_member** (p. 647) operation, committing any changes to the outer object since then.

Some changes to the outer object will not be observable until after you have performed this operation.

If you have called **DDS_DynamicData::bind_complex_member** (p. 647) on a data sample, you must unbind before writing or deleting the sample.

MT Safety:

UNSAFE.

Parameters:

value *<<in>>* *(p. 200)* The same object you passed to **DDS_DynamicData::bind_complex_member** (p. 647). This argument is used for error checking purposes.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::bind_complex_member (p. 647)

6.37.7.17 const DDS_TypeCode* DDS_DynamicData::get_type () const

Get the data type, of which this **DDS_DynamicData** (p. 622) represents an instance.

MT Safety:

UNSAFE.

6.37.7.18 DDS_TCKind DDS_DynamicData::get_type_kind () const

Get the kind of this object's data type.

This is a convenience method. It's equivalent to calling **DDS_DynamicData::get_type** (p. 650) followed by **DDS_TypeCode::kind** (p. 996).

MT Safety:

UNSAFE.

6.37.7.19 DDS_UnsignedLong DDS_DynamicData::get_member_count () const

Get the number of members in this sample.

For objects of type kind **DDS_TK_ARRAY** (p. 66) or **DDS_TK_SEQUENCE** (p. 66), this method returns the number of elements in the collection.

For objects of type kind **DDS_TK_STRUCT** (p. 66) or **DDS_TK_VALUE** (p. 67), it returns the number of fields in the sample, which will always be the same as the number of fields in the type.

For objects of type kind **DDS_TK_SPARSE** (p. 67), it returns the number of fields in the sample, which may be less than or equal to the number of fields in the type.

MT Safety:

UNSAFE.

See also:

DDS_DynamicData::get_member_info_by_index (p. 653)

6.37.7.20 DDS_Boolean DDS_DynamicData::member_exists (const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Indicates whether a member of a particular name/ID exists in this data sample.

Only one of the name and/or ID need by specified.

For objects of type kinds other than **DDS_TK_SPARSE** (p. 67), the result of this method will always be the same as that of **DDS_DynamicData::member_exists_in_type** (p. 652).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

See also:

DDS_DynamicData::member_exists_in_type (p. 652)
DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED (p. 80)

6.37.7.21 DDS_Boolean DDS_DynamicData::member_exists_in_type (const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Indicates whether a member of a particular name/ID exists in this data sample's type.

Only one of the name and/or ID need by specified.

For objects of type kinds other than **DDS_TK_SPARSE** (p. 67), the result of this method will always be the same as that of **DDS_DynamicData::member_exists** (p. 651).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

See also:

DDS_DynamicData::member_exists (p. 651)
DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED (p. 80)

6.37.7.22 DDS_ReturnCode_t DDS_DynamicData::get_member_info (DDS_DynamicDataMemberInfo & *info*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Fill in the given descriptor with information about the identified member of this **DDS_DynamicData** (p. 622) sample.

This operation is valid for objects of **DDS_TCKind** (p. 66) **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67), and **DDS_TK_SPARSE** (p. 67).

MT Safety:

UNSAFE.

Parameters:

info *<<out>> (p. 200)* The descriptor object whose contents will be overwritten by this operations.

member_name *<<in>> (p. 200)* The name of the member for which to get the info or NULL to look up the member by its ID. Only one of the name and the ID may be unspecified.

member_id *<<in>> (p. 200)* The ID of the member for which to get the info, or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::get_member_info_by_index (p. 653)
DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED (p. 80)

6.37.7.23 DDS_ReturnCode_t DDS_DynamicData::get_member_info_by_index (struct DDS_DynamicDataMemberInfo & *info*, DDS_UnsignedLong *index*) const

Fill in the given descriptor with information about the identified member of this **DDS_DynamicData** (p. 622) sample.

This operation is valid for objects of **DDS_TCKind** (p. 66) **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67), and **DDS_TK_SPARSE** (p. 67).

MT Safety:

UNSAFE.

Parameters:

info *<<out>> (p. 200)* The descriptor object whose contents will be overwritten by this operations.

index *<<in>> (p. 200)* The zero-based of the member for which to get the info.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::get_member_info (p. 652)
DDS_DynamicData::get_member_count (p. 650)
DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED (p. 80)

6.37.7.24 DDS_ReturnCode_t DDS_DynamicData::get_member_type (const DDS_TypeCode *& *type_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the type of the given member of this sample.

The member can be looked up either by name or by ID.

This operation is valid for objects of **DDS_TCKind** (p. 66) **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67), and **DDS_TK_SPARSE** (p. 67). For type kinds **DDS_TK_ARRAY** (p. 66) and **DDS_TK_SEQUENCE** (p. 66), the index into the collection is taken to be one less than the ID, if specified. If this index is valid, this operation will return the content type of this collection.

MT Safety:

UNSAFE.

Parameters:

type_out *<<**out**>> (p. 200)* If this method returned success, this argument refers to the found member's type.

member_name *<<**in**>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::get_member_info (p. 652)
DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED (p. 80)

6.37.7.25 DDS_ReturnCode_t DDS_DynamicData::is_member_key (DDS_Boolean & *is_key_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Indicates whether a given member forms part of the key of this sample's data type.

This operation is only valid for samples of types of kind **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67), or **DDS_TK_SPARSE** (p. 67).

Note to users of sparse types: A key member may only have a single representation and is required to exist in every sample.

MT Safety:

UNSAFE.

Parameters:

is_key_out *<<**out**>>* *(p. 200)* If this method returned success, this argument indicates whether the indicated member is part of the key.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

6.37.7.26 DDS_ReturnCode_t DDS_DynamicData::get_long (DDS_Long & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Long** (p. 300) or another type implicitly convertible to it (**DDS_Octet** (p. 299), **DDS_Char** (p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), or **DDS_Enum** (p. 301)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_long (p. 687)

6.37.7.27 DDS_ReturnCode_t DDS_DynamicData::get_short (DDS_Short & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Short** (p. 299) or another type implicitly convertible to it (**DDS_Octet** (p. 299) or **DDS_Char** (p. 299)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_short (p. 687)

6.37.7.28 DDS_ReturnCode_t DDS_DynamicData::get_ulong (DDS_UnsignedLong & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_UnsignedLong** (p. 300) or another type implicitly convertible to it (**DDS_Octet** (p. 299), **DDS_Char**

(p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), or **DDS_Enum** (p. 301)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_ulong (p. 688)

6.37.7.29 DDS_ReturnCode_t DDS_DynamicData::get_ushort (DDS_UnsignedShort & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_UnsignedShort** (p. 299) or another type implicitly convertible to it (**DDS_Octet** (p. 299) or **DDS_Char** (p. 299)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_ushort (p. 688)

6.37.7.30 DDS_ReturnCode_t DDS_DynamicData::get_float (DDS_Float & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Float** (p. 300).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<out>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_float (p. 689)

6.37.7.31 DDS_ReturnCode_t DDS_DynamicData::get_double (DDS_Double & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Double** (p. 300) or another type implicitly convertible to it (**DDS_Float** (p. 300)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_double (p. 690)

6.37.7.32 DDS_ReturnCode_t DDS_DynamicData::get_boolean (DDS_Boolean & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Boolean** (p. 301).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_boolean (p. 690)

6.37.7.33 DDS_ReturnCode_t DDS_DynamicData::get_char (DDS_Char & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Char** (p. 299).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_char (p. 691)

6.37.7.34 DDS_ReturnCode_t DDS_DynamicData::get_octet (DDS_Octet & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Octet** (p. 299).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_octet (p. 691)

6.37.7.35 DDS_ReturnCode_t DDS_DynamicData::get_longlong (DDS_LongLong & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_LongLong** (p. 300) or another type implicitly convertible to it (**DDS_Octet** (p. 299), **DDS_Char** (p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), **DDS_Long** (p. 300), **DDS_UnsignedLong** (p. 300), or **DDS_Enum** (p. 301)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_longlong (p. 692)

6.37.7.36 DDS_ReturnCode_t DDS_DynamicData::get_ulonglong (DDS_UnsignedLongLong & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_UnsignedLongLong** (p. 300) or another type implicitly convertible to it (**DDS_Octet** (p. 299), **DDS_Char** (p. 299), **DDS_Short** (p. 299), **DDS_UnsignedShort** (p. 299), **DDS_Long** (p. 300), **DDS_UnsignedLong** (p. 300), or **DDS_Enum** (p. 301)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>> (p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_ulonglong (p. 693)

6.37.7.37 DDS_ReturnCode_t DDS_DynamicData::get_longdouble (DDS_LongDouble & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_LongDouble** (p. 300) or another type implicitly convertible to it (**DDS_Float** (p. 300) or **DDS_Double** (p. 300)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_longdouble (p. 693)

6.37.7.38 DDS_ReturnCode_t DDS_DynamicData::get_wchar (DDS_Wchar & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Wchar** (p. 299) or another type implicitly convertible to it (**DDS_Char** (p. 299)).

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* If this method returned success, this argument will contain the value of the indicated member.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_wchar (p. 694)

6.37.7.39 DDS_ReturnCode_t DDS_DynamicData::get_string (char *& *value*, DDS_UnsignedLong * *size*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type char*.

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value *<<**out**>>* *(p. 200)* The string into which the middleware should copy the string member. The allocated size of this string is indicated by `size` argument. If the size is sufficient to hold the contents of the member, they will be copied into this string. If `value` is a pointer to NULL, the middleware will allocate a new string for you of sufficient length; it will be your responsibility to free that string. If the size is insufficient but greater than zero, the middleware will *not* free your string; instead, this operation will fail and `size` will contain the minimum required size.

size *<<**inout**>>* *(p. 200)* As an input argument, the allocated size of the string. As an output argument, the actual size of the contents.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_String_alloc (p. 458)
DDS_String_free (p. 459)
DDS_DynamicData::set_string (p. 695)

6.37.7.40 DDS_ReturnCode_t DDS_DynamicData::get_wstring (DDS_Wchar *& *value*, DDS_UnsignedLong * *size*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get the value of the given field, which is of type **DDS_Wchar** (p. 299)*.

The member may be specified by name or by ID.

MT Safety:

UNSAFE.

Parameters:

value *<<out>>* *(p. 200)* The string into which the middleware should copy the string member. The allocated size of this string is indicated by `size` argument. If the size is sufficient to hold the contents of the member, they will be copied into this string. If `value` is a pointer to NULL, the middleware will allocate a new string for you of sufficient length; it will be your responsibility to free that string. If the size is insufficient but greater than zero, the middleware will *not* free your string; instead, this operation will fail and `size` will contain the minimum required size.

size *<<inout>>* *(p. 200)* As an input argument, the allocated size of the string. As an output argument, the actual size of the contents.

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_String_alloc (p. 458)
DDS_String_free (p. 459)
DDS_DynamicData::set_wstring (p. 695)

6.37.7.41 DDS_ReturnCode_t DDS_DynamicData::get_complex_member (DDS_DynamicData & *value_out*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the value of the given field, which is of some composed type.

The member may be of type kind **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67), **DDS_TK_UNION** (p. 66), or **DDS_TK_SPARSE** (p. 67). It may be specified by name or by ID.

This method is logically related to **DDS_DynamicData::bind_complex_member** (p. 647) in that both allow you to examine the state of nested objects. They are different in an important way: this method provides a *copy* of the data; changes to it will not be reflected in the source object.

MT Safety:

UNSAFE.

Parameters:

value_out *<<**out**>>* *(p. 200)* The **DDS_DynamicData** (p. 622) sample whose contents will be overwritten by this operation. This object must *not* be a bound member of another **DDS_DynamicData** (p. 622) sample.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_complex_member (p. 696)
DDS_DynamicData::bind_complex_member (p. 647)

6.37.7.42 DDS_ReturnCode_t DDS_DynamicData::get_long_array (DDS_Long * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_LongSeq** (p. 795).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_long_array (p. 697)
DDS_DynamicData::get_long_seq (p. 677)

6.37.7.43 DDS_ReturnCode_t DDS_DynamicData::get_short_array (DDS_Short * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_ShortSeq** (p. 928).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_short_array (p. 698)
DDS_DynamicData::get_short_seq (p. 678)

6.37.7.44 DDS_ReturnCode_t DDS_DynamicData::get_ulong_array (DDS_UnsignedLong * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_UnsignedLongSeq** (p. 1046).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_ulong_array (p. 699)
DDS_DynamicData::get_ulong_seq (p. 679)

6.37.7.45 DDS_ReturnCode_t DDS_DynamicData::get_ushort_array (DDS_UnsignedShort * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_UnsignedShortSeq** (p. 1047).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_ushort_array (p. 700)
DDS_DynamicData::get_ushort_seq (p. 679)

6.37.7.46 DDS_ReturnCode_t DDS_DynamicData::get_float_array (DDS_Float * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_FloatSeq** (p. 748).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_float_array (p. 700)
DDS_DynamicData::get_float_seq (p. 680)

6.37.7.47 DDS_ReturnCode_t DDS_DynamicData::get_double_array (DDS_Double * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_DoubleSeq** (p. 613).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_double_array (p. 701)
DDS_DynamicData::get_double_seq (p. 681)

6.37.7.48 DDS_ReturnCode_t DDS_DynamicData::get_boolean_array (DDS_Boolean * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_BooleanSeq** (p. 480).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_boolean_array (p. 702)
DDS_DynamicData::get_boolean_seq (p. 682)

6.37.7.49 DDS_ReturnCode_t DDS_DynamicData::get_char_array (DDS_Char * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_CharSeq** (p. 490).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_char_array (p. 703)
DDS_DynamicData::get_char_seq (p. 682)

6.37.7.50 DDS_ReturnCode_t DDS_DynamicData::get_octet_array (DDS_Octet * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_OctetSeq** (p. 801).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_octet_array (p. 703)
DDS_DynamicData::get_octet_seq (p. 683)

6.37.7.51 DDS_ReturnCode_t DDS_DynamicData::get_longlong_array (DDS_LongLong * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_LongLongSeq** (p. 794).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_longlong_array (p. 704)
DDS_DynamicData::get_longlong_seq (p. 684)

6.37.7.52 DDS_ReturnCode_t DDS_DynamicData::get_ulonglong_array (DDS_UnsignedLongLong * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_UnsignedLongLongSeq** (p. 1045).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_ulonglong_array (p. 705)
DDS_DynamicData::get_ulonglong_seq (p. 684)

6.37.7.53 DDS_ReturnCode_t DDS_DynamicData::get_longdouble_array (DDS_LongDouble * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_LongDoubleSeq** (p. 793).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_longdouble_array (p. 706)
DDS_DynamicData::get_longdouble_seq (p. 685)

6.37.7.54 DDS_ReturnCode_t DDS_DynamicData::get_wchar_array (DDS_Wchar * *array*, DDS_UnsignedLong * *length*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given array member.

This method will perform an automatic conversion from **DDS_WcharSeq** (p. 1058).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

array *<<**out**>>* *(p. 200)* An already-allocated array, into which the elements will be copied.

length *<<**inout**>>* *(p. 200)* As an input, the allocated length of `array`. As an output, the number of elements that were copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DynamicData::set_wchar_array (p. 707)
DDS_DynamicData::get_wchar_seq (p. 686)

6.37.7.55 DDS_ReturnCode_t DDS_DynamicData::get_long_seq (DDS_LongSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Long** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_long_seq (p. 707)
DDS_DynamicData::get_long_array (p. 667)

6.37.7.56 DDS_ReturnCode_t DDS_DynamicData::get_short_seq (DDS_ShortSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Short** (p. 299).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_short_seq (p. 708)
DDS_DynamicData::get_short_array (p. 668)

6.37.7.57 DDS_ReturnCode_t DDS_DynamicData::get_ulong_seq (DDS_UnsignedLongSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_UnsignedLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_ulong_seq (p. 709)
DDS_DynamicData::get_ulong_array (p. 668)

6.37.7.58 DDS_ReturnCode_t DDS_DynamicData::get_ushort_seq (DDS_UnsignedShortSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_UnsignedShort** (p. 299).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_ushort_seq (p. 710)
DDS_DynamicData::get_ushort_array (p. 669)

6.37.7.59 DDS_ReturnCode_t DDS_DynamicData::get_float_seq (DDS_FloatSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Long** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_long_seq (p. 707)
DDS_DynamicData::get_long_array (p. 667)

6.37.7.60 DDS_ReturnCode_t DDS_DynamicData::get_double_seq (DDS_DoubleSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Double** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<out>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_double_seq (p. 711)
DDS_DynamicData::get_double_array (p. 671)

6.37.7.61 DDS_ReturnCode_t DDS_DynamicData::get_boolean_seq (DDS_BooleanSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Boolean** (p. 301).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_boolean_seq (p. 712)
DDS_DynamicData::get_boolean_array (p. 672)

6.37.7.62 DDS_ReturnCode_t DDS_DynamicData::get_char_seq (DDS_CharSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Char** (p. 299).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_char_seq (p. 712)
DDS_DynamicData::get_char_array (p. 672)

6.37.7.63 DDS_ReturnCode_t DDS_DynamicData::get_octet_seq (DDS_OctetSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Octet** (p. 299).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_octet_seq (p. 713)
DDS_DynamicData::get_octet_array (p. 673)

6.37.7.64 DDS_ReturnCode_t DDS_DynamicData::get_longlong_seq (DDS_LongLongSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_LongLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_longlong_seq (p. 714)
DDS_DynamicData::get_longlong_array (p. 674)

6.37.7.65 DDS_ReturnCode_t DDS_DynamicData::get_ulonglong_seq (DDS_UnsignedLongLongSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_UnsignedLongLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_ulonglong_seq (p. 715)
DDS_DynamicData::get_ulonglong_array (p. 675)

6.37.7.66 DDS_ReturnCode_t DDS_DynamicData::get_longdouble_seq (DDS_LongDoubleSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_LongDouble** (p. 300).

MT Safety:

UNSAFE.

Parameters:

seq *<<**out**>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_longdouble_seq (p. 715)
DDS_DynamicData::get_longdouble_array (p. 676)

6.37.7.67 DDS_ReturnCode_t DDS_DynamicData::get_wchar_seq (DDS_WcharSeq & *seq*, const char * *member_name*, DDS_DynamicDataMemberId *member_id*) const

Get a copy of the given sequence member.

The provided sequence will be automatically resized as necessary.

This method will perform an automatic conversion from an array of **DDS_Wchar** (p. 299).

MT Safety:

UNSAFE.

Parameters:

seq *<<out>>* *(p. 200)* A sequence, into which the elements will be copied.

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::set_wchar_seq (p. 716)
DDS_DynamicData::get_wchar_array (p. 676)

6.37.7.68 DDS_ReturnCode_t DDS_DynamicData::set_long (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Long *value*)

Set the value of the given field, which is of type **DDS_Long** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_long (p. 655)

6.37.7.69 DDS_ReturnCode_t DDS_DynamicData::set_short (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Short *value*)

Set the value of the given field, which is of type **DDS_Short** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_short (p. 656)

6.37.7.70 DDS_ReturnCode_t DDS_DynamicData::set_ulong (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *value*)

Set the value of the given field, which is of type **DDS_UnsignedLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ulong (p. 656)

6.37.7.71 DDS_ReturnCode_t DDS_DynamicData::set_ushort (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedShort *value*)

Set the value of the given field, which is of type **DDS_UnsignedShort** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ushort (p. 657)

6.37.7.72 DDS_ReturnCode_t DDS_DynamicData::set_float (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Float *value*)

Set the value of the given field, which is of type **DDS_Float** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_float (p. 658)

6.37.7.73 DDS_ReturnCode_t DDS_DynamicData::set_double (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Double *value*)

Set the value of the given field, which is of type **DDS_Double** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_double (p. 658)

6.37.7.74 DDS_ReturnCode_t DDS_DynamicData::set_boolean (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Boolean *value*)

Set the value of the given field, which is of type **DDS_Boolean** (p. 301).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_boolean (p. 659)

6.37.7.75 DDS_ReturnCode_t DDS_DynamicData::set_char (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Char *value*)

Set the value of the given field, which is of type **DDS_Char** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_char (p. 660)

6.37.7.76 DDS_ReturnCode_t DDS_DynamicData::set_octet (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Octet *value*)

Set the value of the given field, which is of type **DDS_Octet** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**in**>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_octet (p. 660)

6.37.7.77 DDS_ReturnCode_t DDS_DynamicData::set_longlong (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_LongLong *value*)

Set the value of the given field, which is of type **DDS_LongLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**in**>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_longlong (p. 661)

6.37.7.78 DDS_ReturnCode_t DDS_DynamicData::set_ulonglong (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLongLong *value*)

Set the value of the given field, which is of type **DDS_UnsignedLongLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ulonglong (p. 662)

6.37.7.79 DDS_ReturnCode_t DDS_DynamicData::set_longdouble (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_LongDouble *value*)

Set the value of the given field, which is of type **DDS_LongDouble** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_longdouble (p. 663)

6.37.7.80 DDS_ReturnCode_t DDS_DynamicData::set_wchar (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_Wchar *value*)

Set the value of the given field, which is of type **DDS_Wchar** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_char (p. 660)

6.37.7.81 DDS_ReturnCode_t DDS_DynamicData::set_string (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const char * *value*)

Set the value of the given field of type char*.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_string (p. 664)

6.37.7.82 DDS_ReturnCode_t DDS_DynamicData::set_wstring (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_Wchar * *value*)

Set the value of the given field of type **DDS_Wchar** (p. 299)*.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>>* *(p. 200)* The value to which to set the member.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_wstring (p. 665)

6.37.7.83 DDS_ReturnCode_t DDS_DynamicData::set_complex_member (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_DynamicData & *value*)

Copy the state of the given **DDS_DynamicData** (p. 622) object into a member of this object.

The member may be of type kind **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67), **DDS_TK_UNION** (p. 66), or **DDS_TK_SPARSE** (p. 67). It may be specified by name or by ID.

Example: Copying Data

This method can be used with **DDS_DynamicData::bind_complex_member** (p. 647) to copy from one **DDS_DynamicData** (p. 622) object to another efficiently. Suppose the following data structure:

```
struct Bar {
     short theShort;
 };

 struct Foo {
     Bar theBar;
 };
```

Support we have two instances of `Foo` (p. 1443): `foo_dst` and `foo_src`. We want to replace the contents of `foo_dst.theBar` with the contents of `foo_src.theBar`. Error handling has been omitted for the sake of brevity.

```
DDS_DynamicData* foo_dst = ...;
 DDS_DynamicData* foo_src = ...;
 DDS_DynamicData* bar = new DDS_DynamicData(NULL, myProperties);
 // Point to the source of the copy:
 foo_src->bind_complex_member(
     "theBar",
     DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED,
```

```
    bar);
// Just one copy:
foo_dst->set_complex_member(
    "theBar",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED,
    bar);
// Tear down:
foo_src->unbind_complex_member(bar);
delete bar;
```

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<in>> (p. 200)* The source **DDS_DynamicData** (p. 622) object whose contents will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_complex_member (p. 666)
DDS_DynamicData::bind_complex_member (p. 647)

6.37.7.84 DDS_ReturnCode_t DDS_DynamicData::set_long_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Long * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_LongSeq** (p. 795).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_long_array (p. 667)
DDS_DynamicData::set_long_seq (p. 707)

6.37.7.85 DDS_ReturnCode_t DDS_DynamicData::set_short_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Short * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_ShortSeq** (p. 928).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_short_array (p. 668)
DDS_DynamicData::set_short_seq (p. 708)

6.37.7.86 DDS_ReturnCode_t DDS_DynamicData::set_ulong_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_UnsignedLong * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_UnsignedLongSeq** (p. 1046).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>> (p. 200)* The length of `array`.

array *<<in>> (p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ulong_array (p. 668)
DDS_DynamicData::set_ulong_seq (p. 709)

6.37.7.87 DDS_ReturnCode_t DDS_DynamicData::set_ushort_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_UnsignedShort * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_UnsignedShortSeq** (p. 1047).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>> (p. 200)* The length of `array`.

array *<<in>> (p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ushort_array (p. 669)
DDS_DynamicData::set_ushort_seq (p. 710)

6.37.7.88 DDS_ReturnCode_t DDS_DynamicData::set_float_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Float * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_FloatSeq** (p. 748).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>> (p. 200)* The length of `array`.

array *<<in>> (p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_float_array (p. 670)
DDS_DynamicData::set_float_seq (p. 710)

6.37.7.89 DDS_ReturnCode_t DDS_DynamicData::set_double_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Double * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_DoubleSeq** (p. 613).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_double_array (p. 671)
DDS_DynamicData::set_double_seq (p. 711)

6.37.7.90 DDS_ReturnCode_t DDS_DynamicData::set_boolean_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Boolean * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_BooleanSeq** (p. 480).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_boolean_array (p. 672)
DDS_DynamicData::set_boolean_seq (p. 712)

6.37.7.91 DDS_ReturnCode_t DDS_DynamicData::set_char_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Char * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_CharSeq** (p. 490).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>> (p. 200)* The length of `array`.

array *<<in>> (p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_char_array (p. 672)
DDS_DynamicData::set_char_seq (p. 712)

6.37.7.92 DDS_ReturnCode_t DDS_DynamicData::set_octet_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Octet * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_OctetSeq** (p. 801).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_octet_array (p. 673)
DDS_DynamicData::set_octet_seq (p. 713)

6.37.7.93 DDS_ReturnCode_t DDS_DynamicData::set_longlong_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_LongLong * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_LongLongSeq** (p. 794).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_longlong_array (p. 674)
DDS_DynamicData::set_longlong_seq (p. 714)

6.37.7.94 DDS_ReturnCode_t DDS_DynamicData::set_ulonglong_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_UnsignedLongLong * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_UnsignedLongLongSeq** (p. 1045).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>>* *(p. 200)* The length of `array`.

array *<<in>>* *(p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ulonglong_array (p. 675)
DDS_DynamicData::set_ulonglong_seq (p. 715)

6.37.7.95 DDS_ReturnCode_t DDS_DynamicData::set_longdouble_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_LongDouble * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_LongDoubleSeq** (p. 793).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>> (p. 200)* The length of `array`.

array *<<in>> (p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_longdouble_array (p. 676)
DDS_DynamicData::set_longdouble_seq (p. 715)

6.37.7.96 DDS_ReturnCode_t DDS_DynamicData::set_wchar_array (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, DDS_UnsignedLong *length*, const DDS_Wchar * *array*)

Set the contents of the given array member.

This method will perform an automatic conversion to **DDS_WcharSeq** (p. 1058).

If the destination array is insufficiently long to store the data, this operation will fail without copying anything.

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

length *<<in>> (p. 200)* The length of `array`.

array *<<in>> (p. 200)* The elements to copy.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_wchar_array (p. 676)
DDS_DynamicData::set_wchar_seq (p. 716)

6.37.7.97 DDS_ReturnCode_t DDS_DynamicData::set_long_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_LongSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Long** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_long_seq (p. 677)
DDS_DynamicData::set_long_array (p. 697)

6.37.7.98 DDS_ReturnCode_t DDS_DynamicData::set_short_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_ShortSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Short** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_short_seq (p. 678)
DDS_DynamicData::set_short_array (p. 698)

6.37.7.99 DDS_ReturnCode_t DDS_DynamicData::set_ulong_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_UnsignedLongSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_UnsignedLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<out>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ulong_seq (p. 679)
DDS_DynamicData::set_ulong_array (p. 699)

6.37.7.100 DDS_ReturnCode_t DDS_DynamicData::set_ushort_seq (const char * member_name, DDS_DynamicDataMemberId member_id, const DDS_UnsignedShortSeq & value)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_UnsignedShort** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<out>> (p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ushort_seq (p. 679)
DDS_DynamicData::set_ushort_array (p. 700)

6.37.7.101 DDS_ReturnCode_t DDS_DynamicData::set_float_seq (const char * member_name, DDS_DynamicDataMemberId member_id, const DDS_FloatSeq & value)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Float** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<out>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_float_seq (p. 680)
DDS_DynamicData::set_float_array (p. 700)

6.37.7.102 DDS_ReturnCode_t DDS_DynamicData::set_double_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_DoubleSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Double** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<out>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_double_seq (p. 681)
DDS_DynamicData::set_double_array (p. 701)

6.37.7.103 DDS_ReturnCode_t DDS_DynamicData::set_boolean_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_BooleanSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Boolean** (p. 301).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<out>> (p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_boolean_seq (p. 682)
DDS_DynamicData::set_boolean_array (p. 702)

6.37.7.104 DDS_ReturnCode_t DDS_DynamicData::set_char_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_CharSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Char** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_char_seq (p. 682)
DDS_DynamicData::set_char_array (p. 703)

6.37.7.105 DDS_ReturnCode_t DDS_DynamicData::set_octet_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_OctetSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Octet** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_octet_seq (p. 683)
DDS_DynamicData::set_octet_array (p. 703)

6.37.7.106 DDS_ReturnCode_t DDS_DynamicData::set_longlong_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_LongLongSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_LongLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_longlong_seq (p. 684)
DDS_DynamicData::set_longlong_array (p. 704)

6.37.7.107 DDS_ReturnCode_t DDS_DynamicData::set_ulonglong_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_UnsignedLongLongSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_UnsignedLongLong** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<in>> (p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<in>> (p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<out>> (p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_ulonglong_seq (p. 684)
DDS_DynamicData::set_ulonglong_array (p. 705)

6.37.7.108 DDS_ReturnCode_t DDS_DynamicData::set_longdouble_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_LongDoubleSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_LongDouble** (p. 300).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_longdouble_seq (p. 685)
DDS_DynamicData::set_longdouble_array (p. 706)

6.37.7.109 DDS_ReturnCode_t DDS_DynamicData::set_wchar_seq (const char * *member_name*, DDS_DynamicDataMemberId *member_id*, const DDS_WcharSeq & *value*)

Set the contents of the given sequence member.

This method will perform an automatic conversion to an array of **DDS_Wchar** (p. 299).

MT Safety:

UNSAFE.

Parameters:

member_name *<<**in**>>* *(p. 200)* The name of the member or NULL to look up the member by its ID.

member_id *<<**in**>>* *(p. 200)* The ID of the member or **DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED** (p. 80) to look up by name. See **Member Names and IDs** (p. 635).

value *<<**out**>>* *(p. 200)* A sequence, from which the elements will be copied.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

See also:

DDS_DynamicData::get_wchar_seq (p. 686)
DDS_DynamicData::set_wchar_array (p. 707)

Get	Set
DDS_DynamicData::get_long (p. 655)	**DDS_DynamicData::set_long** (p. 687)
DDS_DynamicData::get_ulong (p. 656)	**DDS_DynamicData::set_ulong** (p. 688)
DDS_DynamicData::get_short (p. 656)	**DDS_DynamicData::set_short** (p. 687)
DDS_DynamicData::get_ushort (p. 657)	**DDS_DynamicData::set_ushort** (p. 688)
DDS_DynamicData::get_longlong (p. 661)	**DDS_DynamicData::set_longlong** (p. 692)
DDS_DynamicData::get_ulonglong (p. 662)	**DDS_DynamicData::set_ulonglong** (p. 693)
DDS_DynamicData::get_float (p. 658)	**DDS_DynamicData::set_float** (p. 689)
DDS_DynamicData::get_double (p. 658)	**DDS_DynamicData::set_double** (p. 690)
DDS_DynamicData::get_longdouble (p. 663)	**DDS_DynamicData::set_longdouble** (p. 693)
DDS_DynamicData::get_boolean (p. 659)	**DDS_DynamicData::set_boolean** (p. 690)
DDS_DynamicData::get_octet (p. 660)	**DDS_DynamicData::set_octet** (p. 691)
DDS_DynamicData::get_char (p. 660)	**DDS_DynamicData::set_char** (p. 691)
DDS_DynamicData::get_wchar (p. 663)	**DDS_DynamicData::set_wchar** (p. 694)
DDS_DynamicData::get_string (p. 664)	**DDS_DynamicData::set_string** (p. 695)
DDS_DynamicData::get_wstring (p. 665)	**DDS_DynamicData::set_wstring** (p. 695)

Table 6.3: Basic Types

Get	Set
DDS_DynamicData::get_complex_member (p. 666)	**DDS_DynamicData::set_complex_member** (p. 696)

Table 6.4: Structures, Arrays, and Other Complex Types

Get	Set
DDS_DynamicData::get_long_-array (p. 667)	**DDS_DynamicData::set_long_-array** (p. 697)
DDS_DynamicData::get_ulong_-array (p. 668)	**DDS_DynamicData::set_ulong_-array** (p. 699)
DDS_DynamicData::get_short_-array (p. 668)	**DDS_DynamicData::set_short_-array** (p. 698)
DDS_DynamicData::get_-ushort_array (p. 669)	**DDS_DynamicData::set_-ushort_array** (p. 700)
DDS_DynamicData::get_-longlong_array (p. 674)	**DDS_DynamicData::set_-longlong_array** (p. 704)
DDS_DynamicData::get_-ulonglong_array (p. 675)	**DDS_DynamicData::set_-ulonglong_array** (p. 705)
DDS_DynamicData::get_float_-array (p. 670)	**DDS_DynamicData::set_float_-array** (p. 700)
DDS_DynamicData::get_-double_array (p. 671)	**DDS_DynamicData::set_-double_array** (p. 701)
DDS_DynamicData::get_-longdouble_array (p. 676)	**DDS_DynamicData::set_-longdouble_array** (p. 706)
DDS_DynamicData::get_-boolean_array (p. 672)	**DDS_DynamicData::set_-boolean** (p. 690)
DDS_DynamicData::get_octet_-array (p. 673)	**DDS_DynamicData::set_octet_-array** (p. 703)
DDS_DynamicData::get_char_-array (p. 672)	**DDS_DynamicData::set_char_-array** (p. 703)
DDS_DynamicData::get_-wchar_array (p. 676)	**DDS_DynamicData::set_wchar_-array** (p. 707)

Table 6.5: Arrays of Basic Types

Get	Set
DDS_DynamicData::get_long_seq (p. 677)	**DDS_DynamicData::set_long_seq** (p. 707)
DDS_DynamicData::get_ulong_seq (p. 679)	**DDS_DynamicData::set_ulong_seq** (p. 709)
DDS_DynamicData::get_short_seq (p. 678)	**DDS_DynamicData::set_short_seq** (p. 708)
DDS_DynamicData::get_ushort_seq (p. 679)	**DDS_DynamicData::set_ushort_seq** (p. 710)
DDS_DynamicData::get_longlong_seq (p. 684)	**DDS_DynamicData::set_longlong_seq** (p. 714)
DDS_DynamicData::get_ulonglong_seq (p. 684)	**DDS_DynamicData::set_ulonglong_seq** (p. 715)
DDS_DynamicData::get_float_seq (p. 680)	**DDS_DynamicData::set_float_seq** (p. 710)
DDS_DynamicData::get_double_seq (p. 681)	**DDS_DynamicData::set_double_seq** (p. 711)
DDS_DynamicData::get_longdouble_seq (p. 685)	**DDS_DynamicData::set_longdouble_seq** (p. 715)
DDS_DynamicData::get_boolean_seq (p. 682)	**DDS_DynamicData::set_boolean_seq** (p. 712)
DDS_DynamicData::get_octet_seq (p. 683)	**DDS_DynamicData::set_octet_seq** (p. 713)
DDS_DynamicData::get_char_seq (p. 682)	**DDS_DynamicData::set_char_seq** (p. 712)
DDS_DynamicData::get_wchar_seq (p. 686)	**DDS_DynamicData::set_wchar_seq** (p. 716)

Table 6.6: Sequences of Basic Types

6.38 DDS_DynamicDataInfo Struct Reference

A descriptor for a **DDS_DynamicData** (p. 622) object.

Public Attributes

- **DDS_Long member_count**

  *The number of data members in this **DDS_DynamicData** (p. 622) sample.*

- **DDS_Long stored_size**

  *The number of bytes currently used to store the data of this **DDS_DynamicData** (p. 622) sample.*

6.38.1 Detailed Description

A descriptor for a **DDS_DynamicData** (p. 622) object.

See also:

DDS_DynamicData::get_info (p. 646)

6.38.2 Member Data Documentation

6.38.2.1 DDS_Long DDS_DynamicDataInfo::member_count

The number of data members in this **DDS_DynamicData** (p. 622) sample.

6.38.2.2 DDS_Long DDS_DynamicDataInfo::stored_size

The number of bytes currently used to store the data of this **DDS_DynamicData** (p. 622) sample.

6.39 DDS_DynamicDataMemberInfo Struct Reference

A descriptor for a single member (i.e. field) of dynamically defined data type.

Public Attributes

- **DDS_DynamicDataMemberId member_id**

  *An integer that uniquely identifies the data member within this **DDS_DynamicData** (p. 622) sample's type.*

- const char * **member_name**

  The string name of the data member.

- **DDS_Boolean member_exists**

  Indicates whether the corresponding member of the data type actually exists in this sample.

- **DDS_TCKind member_kind**

  The kind of type of this data member (e.g. integer, structure, etc.).

- **DDS_UnsignedLong element_count**

  The number of elements within this data member.

- **DDS_TCKind element_kind**

  The kind of type of the elements within this data member.

6.39.1 Detailed Description

A descriptor for a single member (i.e. field) of dynamically defined data type.

See also:

DDS_DynamicData::get_member_info (p. 652)

6.39.2 Member Data Documentation

6.39.2.1 DDS_DynamicDataMemberId DDS_DynamicDataMemberInfo::member_id

An integer that uniquely identifies the data member within this **DDS_DynamicData** (p. 622) sample's type.

For sparse data types, this value will be assigned by the type designer. For types defined in IDL, it will be assigned automatically by the middleware based on the member's declaration order within the type.

See also:

DDS_TCKind (p. 66)

6.39.2.2 const char* DDS_DynamicDataMemberInfo::member_name

The string name of the data member.

This name will be unique among members of the same type. However, a single named member may have multiple type representations.

See also:

DDS_DynamicDataMemberInfo::representation_count

6.39.2.3 DDS_Boolean DDS_DynamicDataMemberInfo::member_exists

Indicates whether the corresponding member of the data type actually exists in this sample.

For non-sparse data types, this value will always be **DDS_BOOLEAN_TRUE** (p. 298).

See also:

DDS_TCKind (p. 66)

6.39.2.4 DDS_TCKind DDS_DynamicDataMemberInfo::member_kind

The kind of type of this data member (e.g. integer, structure, etc.).

This is a convenience field; it is equivalent to looking up the member in the **DDS_TypeCode** (p. 992) and getting the **DDS_TCKind** (p. 66) from there.

6.39.2.5 DDS_UnsignedLong DDS_DynamicDataMemberInfo::element_count

The number of elements within this data member.

This information is only valid for members of array or sequence types. Members of other types will always report zero (0) here.

6.39.2.6 DDS_TCKind DDS_DynamicDataMemberInfo::element_kind

The kind of type of the elements within this data member.

This information is only valid for members of array or sequence types. Members of other types will always report **DDS_TK_NULL** (p. 66) here.

6.40 DDS_DynamicDataProperty_t Struct Reference

A collection of attributes used to configure **DDS_DynamicData** (p. 622) objects.

Public Attributes

- **DDS_Long buffer_initial_size**

  *The initial amount of memory used by this **DDS_DynamicData** (p. 622) object, in bytes.*

- **DDS_Long buffer_max_size**

  *The maximum amount of memory that this **DDS_DynamicData** (p. 622) object may use, in bytes.*

6.40.1 Detailed Description

A collection of attributes used to configure **DDS_DynamicData** (p. 622) objects.

6.40.2 Member Data Documentation

6.40.2.1 DDS_Long DDS_DynamicDataProperty_t::buffer_initial_size

The initial amount of memory used by this **DDS_DynamicData** (p. 622) object, in bytes.

See also:

DDS_DynamicDataProperty_t::buffer_max_size (p. 725)

6.40.2.2 DDS_Long DDS_DynamicDataProperty_t::buffer_max_size

The maximum amount of memory that this **DDS_DynamicData** (p. 622) object may use, in bytes.

It will grow to this size from the initial size as needed.

See also:

DDS_DynamicDataProperty_t::buffer_initial_size (p. 725)

6.41 DDS_DynamicDataSeq Struct Reference

An ordered collection of **DDS_DynamicData** (p. 622) elements.

6.41.1 Detailed Description

An ordered collection of **DDS_DynamicData** (p. 622) elements.

Instantiates `FooSeq` (p. 1494) < **DDS_DynamicData** (p. 622) > .

See also:

FooSeq (p. 1494)
DDS_DynamicData (p. 622)

6.42 DDS_DynamicDataTypeProperty_t Struct Reference

A collection of attributes used to configure **DDSDynamicDataTypeSupport** (p. 1246) objects.

Public Attributes

- struct **DDS_DynamicDataProperty_t data**

  *These properties will be provided to every new **DDS_DynamicData** (p. 622) sample created from the **DDSDynamicDataTypeSupport** (p. 1246).*

- struct **DDS_DynamicDataTypeSerializationProperty_t serialization**

  Properties that govern how the data of this type will be serialized on the network.

6.42.1 Detailed Description

A collection of attributes used to configure **DDSDynamicDataTypeSupport** (p. 1246) objects.

The properties of a **DDSDynamicDataTypeSupport** (p. 1246) object contain the properties that will be used to instantiate any samples created by that object.

6.42.2 Member Data Documentation

6.42.2.1 struct DDS_DynamicDataProperty_t DDS_DynamicDataTypeProperty_t::data [read]

These properties will be provided to every new **DDS_DynamicData** (p. 622) sample created from the **DDSDynamicDataTypeSupport** (p. 1246).

6.42.2.2 struct DDS_DynamicDataTypeSerializationProperty_t DDS_DynamicDataTypeProperty_t::serialization [read]

Properties that govern how the data of this type will be serialized on the network.

6.43 DDS_DynamicDataTypeSerializationProperty_t Struct Reference

Properties that govern how data of a certain type will be serialized on the network.

Public Attributes

- **DDS_Boolean use_42e_compatible_alignment**

  Use RTI Connext 4.2e-compatible alignment for large primitive types.

- **DDS_UnsignedLong max_size_serialized**

  The maximum number of bytes that objects of a given type could consume when serialized on the network.

6.43.1 Detailed Description

Properties that govern how data of a certain type will be serialized on the network.

6.43.2 Member Data Documentation

6.43.2.1 DDS_Boolean DDS_DynamicDataTypeSerializationProperty_t::use_42e_compatible_alignment

Use RTI Connext 4.2e-compatible alignment for large primitive types.

In RTI Connext 4.2e, the default alignment for large primitive types – **DDS_LongLong** (p. 300), **DDS_UnsignedLongLong** (p. 300), **DDS_Double** (p. 300), and **DDS_LongDouble** (p. 300) – was not RTPS-compliant. This compatibility mode allows applications targeting post-4.2e versions of RTI Connext to interoperate with 4.2e-based applications, regardless of the data types they use.

If this flag is not set, all data will be serialized in an RTPS-compliant manner, which for the types listed above, will not be interoperable with RTI Connext 4.2e.

6.43.2.2 DDS_UnsignedLong DDS_DynamicDataTypeSerializationProperty_t::max_size_serialized

The maximum number of bytes that objects of a given type could consume when serialized on the network.

This value is used to set the sizes of certain internal middleware buffers.

The effective value of the maximum serialized size will be the value of this field or the size automatically inferred from the type's **DDS_TypeCode** (p. 992), whichever is smaller.

6.44 DDS_EndpointGroup_t Struct Reference

Specifies a group of endpoints that can be collectively identified by a name and satisfied by a quorum.

Public Attributes

- char * **role_name**

  Defines the role name of the endpoint group.

- int **quorum_count**

  Defines the minimum number of members that satisfies the endpoint group.

6.44.1 Detailed Description

Specifies a group of endpoints that can be collectively identified by a name and satisfied by a quorum.

6.44.2 Member Data Documentation

6.44.2.1 char* DDS_EndpointGroup_t::role_name

Defines the role name of the endpoint group.

If used in the **DDS_AvailabilityQosPolicy** (p. 471) on a **DDSDataWriter** (p. 1113), it specifies the name that identifies a Durable Subcription.

6.44.2.2 int DDS_EndpointGroup_t::quorum_count

Defines the minimum number of members that satisfies the endpoint group.

If used in the **DDS_AvailabilityQosPolicy** (p. 471) on a **DDSDataWriter** (p. 1113), it specifies the number of DataReaders that must acknowledge a sample before the sample is considered to be acknowledged by the Durable Subscription.

6.45 DDS_EndpointGroupSeq Struct Reference

A sequence of **DDS_EndpointGroup_t** (p. 731).

6.45.1 Detailed Description

A sequence of **DDS_EndpointGroup_t** (p. 731).

In the context of Collaborative DataWriters, it can be used by a **DDS-DataReader** (p. 1087) to define a group of remote DataWriters that the **DDS-DataReader** (p. 1087) will wait to discover before skipping missing samples.

In the context of Durable Subscriptions, it can be used to create a set of Durable Subscriptions identified by a name and a quorum count.

Instantiates:

<<generic>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_EndpointGroup_t (p. 731)

6.46 DDS_EntityFactoryQosPolicy Struct Reference

A QoS policy for all **DDSEntity** (p. 1253) types that can act as factories for one or more other **DDSEntity** (p. 1253) types.

Public Attributes

- **DDS_Boolean autoenable_created_entities**

  Specifies whether the entity acting as a factory automatically enables the instances it creates.

6.46.1 Detailed Description

A QoS policy for all **DDSEntity** (p. 1253) types that can act as factories for one or more other **DDSEntity** (p. 1253) types.

Entity:

DDSDomainParticipantFactory (p. 1216), **DDSDomainParticipant** (p. 1139), **DDSPublisher** (p. 1346), **DDSSubscriber** (p. 1390)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **YES** (p. 340)

6.46.2 Usage

This policy controls the behavior of the **DDSEntity** (p. 1253) as a factory for other entities. It controls whether or not child entities are created in the enabled state.

RTI Connext uses a factory design pattern for creating DDS Entities. That is, a parent entity must be used to create child entities. DomainParticipants create Topics, Publishers and Subscribers. Publishers create DataWriters. Subscribers create DataReaders.

By default, a child object is enabled upon creation (initialized and may be actively used). With this QoS policy, a child object can be created in a disabled state. A disabled entity is only partially initialized and cannot be used until the entity is enabled. Note: an entity can only be *enabled*; it cannot be *disabled* after it has been enabled.

This QoS policy is useful to synchronize the initialization of DDS Entities. For example, when a **DDSDataReader** (p. 1087) is created in an enabled state, its existence is immediately propagated for discovery and the **DDSDataReader** (p. 1087) object's listener called as soon as data is received. The initialization process for an application may extend beyond the creation of the **DDSDataReader** (p. 1087), and thus, it may not be desireable for the **DDSDataReader** (p. 1087) to start to receive or process any data until the initialization process is complete. So by creating readers in a disabled state, your application can make sure that no data is received until the rest of the application initialization is complete, and at that time, enable the them.

Note: if an entity is disabled, then all of the child entities it creates will be disabled too, regardless of the setting of this QoS policy. However, enabling a disabled entity will enable all of its children if this QoS policy is set to automatically enable children entities.

This policy is mutable. A change in the policy affects only the entities created after the change, not any previously created entities.

6.46.3 Member Data Documentation

6.46.3.1 DDS_Boolean DDS_EntityFactoryQosPolicy::autoenable_created_entities

Specifies whether the entity acting as a factory automatically enables the instances it creates.

The setting of `autoenable_created_entities` to **DDS_BOOLEAN_TRUE** (p. 298) indicates that the factory `create_<entity>` operation(s) will automatically invoke the **DDSEntity::enable** (p. 1256) operation each time a new **DDSEntity** (p. 1253) is created. Therefore, the **DDSEntity** (p. 1253) returned by `create_<entity>` will already be enabled. A setting of **DDS_BOOLEAN_FALSE** (p. 299) indicates that the **DDSEntity** (p. 1253) will not be automatically enabled. Your application will need to call **DDSEntity::enable** (p. 1256) itself.

The default setting of `autoenable_created_entities` = **DDS_BOOLEAN_TRUE** (p. 298) means that, by default, it is not necessary to explicitly call **DDSEntity::enable** (p. 1256) on newly created entities.

[default] DDS_BOOLEAN_TRUE (p. 298)

6.47 DDS_EntityNameQosPolicy Struct Reference

Assigns a name and a role name to a **DDSDomainParticipant** (p. 1139), **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087). These names will be visible during the discovery process and in RTI tools to help you visualize and debug your system.

Public Attributes

- char * **name**

  The name of the entity.

- char * **role_name**

  The entity role name.

6.47.1 Detailed Description

Assigns a name and a role name to a **DDSDomainParticipant** (p. 1139), **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087). These names will be visible during the discovery process and in RTI tools to help you visualize and debug your system.

Entity:

DDSDomainParticipant (p. 1139), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = NO;
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.47.2 Usage

The name and role name can only be 255 characters in length.

The strings must be null-terminated strings allocated with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459).

If you provide a non-null pointer when getting the QoS, then it should point to valid memory that can be written to, to avoid ungraceful failures.

6.47.3 Member Data Documentation

6.47.3.1 char* DDS_EntityNameQosPolicy::name

The name of the entity.

[default] "[ENTITY]" for **DDSDomainParticipant** (p. 1139). null for **DDSDataReader** (p. 1087) and **DDSDataWriter** (p. 1113)

[range] Null terminated string with length not exceeding 255. It can be null.

6.47.3.2 char* DDS_EntityNameQosPolicy::role_name

The entity role name.

With Durable Subscriptions this name is used to specify to which Durable Subscription the **DDSDataReader** (p. 1087) belongs.

With Collaborative DataWriters this name is used to specify to which endpoint group the **DDSDataWriter** (p. 1113) belongs.

[range] Null terminated string with length not exceeding 255. It can be null.

[default] null

6.48 DDS_EnumMember Struct Reference

A description of a member of an enumeration.

Public Attributes

- char * **name**

  The name of the enumeration member.

- **DDS_Long ordinal**

  The value associated the the enumeration member.

6.48.1 Detailed Description

A description of a member of an enumeration.

See also:

DDS_EnumMemberSeq (p. 738)
DDS_TypeCodeFactory::create_enum_tc (p. 1032)

6.48.2 Member Data Documentation

6.48.2.1 char* DDS_EnumMember::name

The name of the enumeration member.

Cannot be NULL.

6.48.2.2 DDS_Long DDS_EnumMember::ordinal

The value associated the the enumeration member.

6.49 DDS_EnumMemberSeq Struct Reference

Defines a sequence of enumerator members.

6.49.1 Detailed Description

Defines a sequence of enumerator members.

See also:

DDS_EnumMember (p. 737)
FooSeq (p. 1494)
DDS_TypeCodeFactory::create_enum_tc (p. 1032)

6.50 DDS_EventQosPolicy Struct Reference

Settings for event.

Public Attributes

- struct **DDS_ThreadSettings_t thread**

  Event thread QoS.

- **DDS_Long initial_count**

  The initial number of events.

- **DDS_Long max_count**

  The maximum number of events.

6.50.1 Detailed Description

Settings for event.

In a **DDSDomainParticipant** (p. 1139), a thread is dedicated to handle all timed events, including checking for timeouts and deadlines and executing internal and user-defined timeout or exception handling routines/callbacks.

This QoS policy allows you to configure thread properties such as priority level and stack size. You can also configure the maximum number of events that can be posted to the event thread. By default, a **DDSDomainParticipant** (p. 1139) will dynamically allocate memory as needed for events posted to the event thread. However, by setting a maximum value or setting the initial and maximum value to be the same, you can either bound the amount of memory allocated for the event thread or prevent a **DDSDomainParticipant** (p. 1139) from dynamically allocating memory for the event thread after initialization.

This QoS policy is an extension to the DDS standard.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.50.2 Member Data Documentation

6.50.2.1 struct DDS_ThreadSettings_t DDS_EventQosPolicy::thread [read]

Event thread QoS.

There is only one event thread.

Priority:

[default] The actual value depends on your architecture:

For Windows: -2

For Solaris: OS default priority

For Linux: OS default priority

For LynxOS: 13

For INTEGRITY: 80

For VxWorks: 110

For all others: OS default priority.

Stack Size:

[default] The actual value depends on your architecture:

For Windows: OS default stack size

For Solaris: OS default stack size

For Linux: OS default stack size

For LynxOS: 4∗16∗1024

For INTEGRITY: 4∗20∗1024

For VxWorks: 4∗16∗1024

For all others: OS default stack size.

Mask:

[default] mask = **DDS_THREAD_SETTINGS_FLOATING_POINT** (p. 329) | **DDS_THREAD_SETTINGS_STDIO** (p. 329)

6.50.2.2 DDS_Long DDS_EventQosPolicy::initial_count

The initial number of events.

[default] 256

[range] [1, 1 million], <= max_count

6.50.2.3 DDS_Long DDS_EventQosPolicy::max_count

The maximum number of events.

The maximum number of events. If the limit is reached, no new event can be added.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_count

6.51 DDS_ExclusiveAreaQosPolicy Struct Reference

Configures multi-thread concurrency and deadlock prevention capabilities.

Public Attributes

- **DDS_Boolean use_shared_exclusive_area**

  *Whether the **DDSEntity** (p. 1253) is protected by its own exclusive area or the shared exclusive area.*

6.51.1 Detailed Description

Configures multi-thread concurrency and deadlock prevention capabilities.

An "exclusive area" is an abstraction of a multi-thread-safe region. Each entity is protected by one and only one exclusive area, although a single exclusive area may be shared by multiple entities.

Conceptually, an exclusive area is a mutex or monitor with additional deadlock protection features. If a **DDSEntity** (p. 1253) has "entered" its exclusive area to perform a protected operation, no other **DDSEntity** (p. 1253) sharing the same exclusive area may enter it until the first **DDSEntity** (p. 1253) "exits" the exclusive area.

Entity:

DDSPublisher (p. 1346), **DDSSubscriber** (p. 1390)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

See also:

DDSListener (p. 1318)

6.51.2 Usage

Exclusive Areas (EAs) allow RTI Connext to be multi-threaded while preventing deadlock in multi-threaded applications. EAs prevent a **DDSDomainParticipant** (p. 1139) object's internal threads from deadlocking with each other when

executing internal code as well as when executing the code of user-registered listener callbacks.

Within an EA, all calls to the code protected by the EA are single threaded. Each **DDSDomainParticipant** (p. 1139), **DDSPublisher** (p. 1346) and **DDSSubscriber** (p. 1390) entity represents a separate EA. Thus all DataWriters of the same Publisher and all DataReaders of the same Subscriber share the EA of its parent. Note: this means that operations on the DataWriters of the same Publisher and on the DataReaders of the same Subscriber will be serialized, even when invoked from multiple concurrent application threads.

Within an EA, there are limitations on how code protected by a different EA can be accessed. For example, when received data is being processed by user code in the DataReader Listener, within a Subscriber EA, the user code may call the **FooDataWriter::write** (p. 1484) operation of a DataWriter that is protected by the EA of its Publisher, so you can send data in the function called to process received data. However, you cannot create entities or call functions that are protected by the EA of the **DDSDomainParticipant** (p. 1139). See Chapter 4 in the `User's Manual` for complete documentation on Exclusive Areas.

With this QoS policy, you can force a **DDSPublisher** (p. 1346) or **DDSSubscriber** (p. 1390) to share the same EA as its **DDSDomainParticipant** (p. 1139). Using this capability, the restriction of not being able to create entities in a DataReader Listener's on_data_available() callback is lifted. However, the tradeoff is that the application has reduced concurrency through the Entities that share an EA.

Note that the restrictions on calling methods in a different EA only exist for user code that is called in registered DDS Listeners by internal DomainParticipant threads. User code may call all RTI Connext functions for any DDS Entities from their own threads at any time.

6.51.3 Member Data Documentation

6.51.3.1 DDS_Boolean DDS_ExclusiveAreaQosPolicy::use_shared_exclusive_area

Whether the **DDSEntity** (p. 1253) is protected by its own exclusive area or the shared exclusive area.

All writers belonging to the same **DDSPublisher** (p. 1346) are protected by the same exclusive area as the **DDSPublisher** (p. 1346) itself. The same is true of all readers belonging to the same **DDSSubscriber** (p. 1390). Typically, the publishers and subscribers themselves do not share their exclusive areas with each other; each has its own. This configuration maximizes the concurrency of the system because independent readers and writers do not need to take the same mutexes in order to operate. However, it places some restrictions on the

operations that may be invoked from within listener callbacks because of the possibility of a deadlock. See the **DDSListener** (p. 1318) documentation for more details.

If this field is set to **DDS_BOOLEAN_FALSE** (p. 299), the default more concurrent behavior will be used. In the event that this behavior is insufficiently flexible for your application, you may set this value to **DDS_BOOLEAN_TRUE** (p. 298). In that case, the **DDSSubscriber** (p. 1390) or **DDSPublisher** (p. 1346) in question, and all of the readers or writers (as appropriate) created from it, will share a global exclusive area. This global exclusive area is shared by all entities whose value for this QoS field is **DDS_BOOLEAN_TRUE** (p. 298). By sharing the same exclusive area across a larger number of entities, the concurrency of the system will be decreased; however, some of the callback restrictions will be relaxed.

[default] DDS_BOOLEAN_FALSE (p. 299)

6.52 DDS_ExpressionProperty Struct Reference

Public Attributes

- DDS_Boolean key_only_filter
- DDS_Boolean writer_side_filter_optimization

6.52.1 Detailed Description

6.52.2 Member Data Documentation

6.52.2.1 DDS_Boolean DDS_ExpressionProperty::key_only_filter

6.52.2.2 DDS_Boolean DDS_ExpressionProperty::writer_side_filter_optimization

6.53 DDS_FactoryPluginSupport Struct Reference

Interface for creating and manipulating DDS entities.

6.53.1 Detailed Description

Interface for creating and manipulating DDS entities.

The FactoryPluginSupport is an interface for creating DDS entities as well as others elements from different layers as C++, Java, etc.

The interface must be implemented by using the functions provided for the corresponding layer, that is, for the C layer using C functions, for the C++ layer using C++ functions, and so forth.

6.54 DDS_FilterSampleInfo Struct Reference

Public Attributes

- struct **DDS_SampleIdentity_t related_sample_identity**
- **DDS_Long priority**

6.54.1 Detailed Description

6.54.2 Member Data Documentation

6.54.2.1 struct DDS_SampleIdentity_t DDS_FilterSampleInfo::related_sample_identity [read]

6.54.2.2 DDS_Long DDS_FilterSampleInfo::priority

6.55 DDS_FloatSeq Struct Reference

Instantiates FooSeq (p. 1494) < **DDS_Float** (p. 300) >.

6.55.1 Detailed Description

Instantiates FooSeq (p. 1494) < **DDS_Float** (p. 300) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Float (p. 300)
FooSeq (p. 1494)

6.56 DDS_FlowControllerProperty_t Struct Reference

Determines the flow control characteristics of the **DDSFlowController** (p. 1259).

Public Attributes

- **DDS_FlowControllerSchedulingPolicy scheduling_policy**

  Scheduling policy.

- struct **DDS_FlowControllerTokenBucketProperty_t token_bucket**

  Settings for the token bucket.

6.56.1 Detailed Description

Determines the flow control characteristics of the **DDSFlowController** (p. 1259).

The flow control characteristics shape the network traffic by determining how often and in what order associated asynchronous **DDSDataWriter** (p. 1113) instances are serviced and how much data they are allowed to send.

Note that these settings apply directly to the **DDSFlowController** (p. 1259), and do not depend on the number of **DDSDataWriter** (p. 1113) instances the **DDSFlowController** (p. 1259) is servicing. For instance, the specified flow rate does *not* double simply because two **DDSDataWriter** (p. 1113) instances are waiting to write.

Entity:

DDSFlowController (p. 1259)

Properties:

RxO (p. 340) = N/A

Changeable (p. 340) = **NO** (p. 340) for **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750), **YES** (p. 340) for **DDS_FlowControllerProperty_t::token_bucket** (p. 750). However, the special value of **DDS_DURATION_INFINITE** (p. 305) as **DDS_FlowControllerTokenBucketProperty_t::period** (p. 753) is strictly

used to create an *on-demand* **DDSFlowController** (p. 1259). The token period cannot toggle from an infinite to finite value (or vice versa). It can, however, change from one finite value to another.

6.56.2 Member Data Documentation

6.56.2.1 DDS_FlowControllerSchedulingPolicy DDS_FlowControllerProperty_t::scheduling_policy

Scheduling policy.

Determines the scheduling policy for servicing the **DDSDataWriter** (p. 1113) instances associated with the **DDSFlowController** (p. 1259).

[default] idref_FlowControllerSchedulingPolicy_EDF_FLOW_CONTROLLER_SCHED_POLICY

6.56.2.2 struct DDS_FlowControllerTokenBucketProperty_t DDS_FlowControllerProperty_t::token_bucket [read]

Settings for the token bucket.

6.57 DDS_FlowControllerTokenBucketProperty_t Struct Reference

DDSFlowController (p. 1259) uses the popular token bucket approach for open loop network flow control. The flow control characteristics are determined by the token bucket properties.

Public Attributes

- **DDS_Long max_tokens**

  Maximum number of tokens than can accumulate in the token bucket.

- **DDS_Long tokens_added_per_period**

  The number of tokens added to the token bucket per specified period.

- **DDS_Long tokens_leaked_per_period**

  The number of tokens removed from the token bucket per specified period.

- struct **DDS_Duration_t period**

  Period for adding tokens to and removing tokens from the bucket.

- **DDS_Long bytes_per_token**

  Maximum number of bytes allowed to send for each token available.

6.57.1 Detailed Description

DDSFlowController (p. 1259) uses the popular token bucket approach for open loop network flow control. The flow control characteristics are determined by the token bucket properties.

Asynchronously published samples are queued up and transmitted based on the token bucket flow control scheme. The token bucket contains tokens, each of which represents a number of bytes. Samples can be sent only when there are sufficient tokens in the bucket. As samples are sent, tokens are consumed. The number of tokens consumed is proportional to the size of the data being sent. Tokens are replenished on a periodic basis.

The rate at which tokens become available and other token bucket properties determine the network traffic flow.

Note that if the same sample must be sent to multiple destinations, separate tokens are required for each destination. Only when multiple samples are destined to the same destination will they be co-alesced and sent using the same

token(s). In other words, each token can only contribute to a single network packet.

Entity:

DDSFlowController (p. 1259)

Properties:

RxO (p. 340) = N/A

Changeable (p. 340) = **YES** (p. 340). However, the special value of **DDS_DURATION_INFINITE** (p. 305) as **DDS_FlowControllerTokenBucketProperty_t::period** (p. 753) is strictly used to create an *on-demand* **DDSFlowController** (p. 1259). The token period cannot toggle from an infinite to finite value (or vice versa). It can, however, change from one finite value to another.

6.57.2 Member Data Documentation

6.57.2.1 DDS_Long DDS_FlowControllerTokenBucketProperty_t::max_tokens

Maximum number of tokens than can accumulate in the token bucket.

The number of tokens in the bucket will never exceed this value. Any excess tokens are discarded. This property value, combined with **DDS_FlowControllerTokenBucketProperty_t::bytes_per_token** (p. 754), determines the maximum allowable data burst.

Use **DDS_LENGTH_UNLIMITED** (p. 371) to allow accumulation of an unlimited amount of tokens (and therefore potentially an unlimited burst size).

[default] DDS_LENGTH_UNLIMITED (p. 371)

6.57.2.2 DDS_Long DDS_FlowControllerTokenBucketProperty_t::tokens_added_per_period

The number of tokens added to the token bucket per specified period.

DDSFlowController (p. 1259) transmits data only when tokens are available. Tokens are periodically replenished. This field determines the number of tokens added to the token bucket with each periodic replenishment.

Available tokens are distributed to associated **DDSDataWriter** (p. 1113) instances based on the **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750).

Use **DDS_LENGTH_UNLIMITED** (p. 371) to add the maximum number of tokens allowed by **DDS_FlowControllerTokenBucketProperty_t::max_tokens** (p. 752).

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

6.57.2.3 DDS_Long DDS_FlowControllerTokenBucketProperty_t::tokens_leaked_per_period

The number of tokens removed from the token bucket per specified period.

DDSFlowController (p. 1259) transmits data only when tokens are available. When tokens are replenished and there are sufficient tokens to send all samples in the queue, this property determines whether any or all of the leftover tokens remain in the bucket.

Use **DDS_LENGTH_UNLIMITED** (p. 371) to remove all excess tokens from the token bucket once all samples have been sent. In other words, no token accumulation is allowed. When new samples are written after tokens were purged, the earliest point in time at which they can be sent is at the next periodic replenishment.

[**default**] 0

6.57.2.4 struct DDS_Duration_t DDS_FlowControllerTokenBucketProperty_t::period [read]

Period for adding tokens to and removing tokens from the bucket.

DDSFlowController (p. 1259) transmits data only when tokens are available. This field determines the period by which tokens are added or removed from the token bucket.

The special value **DDS_DURATION_INFINITE** (p. 305) can be used to create an *on-demand* **DDSFlowController** (p. 1259), for which tokens are no longer replenished periodically. Instead, tokens must be added explicitly by calling **DDSFlowController::trigger_flow** (p. 1261). This external trigger adds **DDS_FlowControllerTokenBucketProperty_t::tokens_added_per_period** (p. 752) tokens each time it is called (subject to the other property settings).

[**default**] 1 second

[**range**] [0,1 year] or **DDS_DURATION_INFINITE** (p. 305)

6.57.2.5 DDS_Long DDS_FlowControllerTokenBucketProperty_t::bytes_per_token

Maximum number of bytes allowed to send for each token available.

DDSFlowController (p. 1259) transmits data only when tokens are available. This field determines the number of bytes that can actually be transmitted based on the number of tokens.

Tokens are always consumed in whole by each **DDSDataWriter** (p. 1113). That is, in cases where **DDS_FlowControllerTokenBucketProperty_t::bytes_per_token** (p. 754) is greater than the sample size, multiple samples may be sent to the same destination using a single token (regardless of **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750)).

Where fragmentation is required, the fragment size will be **DDS_FlowControllerTokenBucketProperty_t::bytes_per_token** (p. 754) or the minimum largest message size across all transports installed with the **DDSDataWriter** (p. 1113), whichever is less.

Use **DDS_LENGTH_UNLIMITED** (p. 371) to indicate that an unlimited number of bytes can be transmitted per token. In other words, a single token allows the recipient **DDSDataWriter** (p. 1113) to transmit all its queued samples to a single destination. A separate token is required to send to each additional destination.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1024,**DDS_LENGTH_UNLIMITED** (p. 371)]

6.58 DDS_GroupDataQosPolicy Struct Reference

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Public Attributes

- struct **DDS_OctetSeq value**

  a sequence of octets

6.58.1 Detailed Description

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Entity:

DDSPublisher (p. 1346), **DDSSubscriber** (p. 1390)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **YES** (p. 340)

See also:

DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.58.2 Usage

The additional information is attached to a **DDSPublisher** (p. 1346) or **DDSSubscriber** (p. 1390). This extra data is not used by RTI Connext itself. When a remote application discovers the **DDSPublisher** (p. 1346) or **DDSSubscriber** (p. 1390), it can access that information and use it for its own purposes.

Use cases for this QoS policy, as well as the **DDS_TopicDataQosPolicy** (p. 963) and **DDS_UserDataQosPolicy** (p. 1048), are often application-to-application identification, authentication, authorization, and encryption purposes. For example, applications can use Group or User Data to send security certificates to each other for RSA-type security.

In combination with **DDSDataReaderListener** (p. 1108), **DDSDataWriterListener** (p. 1133) and operations such as **DDSDomainParticipant::ignore_publication** (p. 1189) and **DDSDomainParticipant::ignore_subscription** (p. 1190), this QoS policy can help an application to define and enforce its own security policies. For example, an application can implement matching policies similar to those of the **DDS_PartitionQosPolicy** (p. 820), except the decision can be made based on an application-defined policy.

The use of this QoS is not limited to security; it offers a simple, yet flexible extensibility mechanism.

Important: RTI Connext stores the data placed in this policy in pre-allocated pools. It is therefore necessary to configure RTI Connext with the maximum size of the data that will be stored in policies of this type. This size is configured with **DDS_DomainParticipantResourceLimitsQosPolicy::publisher_group_data_max_length** (p. 606) and **DDS_DomainParticipantResourceLimitsQosPolicy::subscriber_group_data_max_length** (p. 607).

6.58.3 Member Data Documentation

6.58.3.1 struct DDS_OctetSeq DDS_GroupDataQosPolicy::value [read]

a sequence of octets

[default] Empty (zero-sized)

[range] Octet sequence of length [0,max_length]

6.59 DDS_GUID_t Struct Reference

Type for *GUID* (Global Unique Identifier) representation.

Public Attributes

- **DDS_Octet value** [16]
  A 16 byte array containing the GUID value.

6.59.1 Detailed Description

Type for *GUID* (Global Unique Identifier) representation.

Represents a 128 bit GUID.

6.59.2 Member Data Documentation

6.59.2.1 DDS_Octet DDS_GUID_t::value[16]

A 16 byte array containing the GUID value.

6.60 DDS_HistoryQosPolicy Struct Reference

Specifies the behavior of RTI Connext in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers.

Public Attributes

- **DDS_HistoryQosPolicyKind kind**

  Specifies the kind of history to be kept.

- **DDS_Long depth**

  Specifies the number of samples to be kept, when the `kind` *is* ***DDS_KEEP_LAST_HISTORY_QOS*** *(p. 368).*

- **DDS_RefilterQosPolicyKind refilter**

  <<**eXtension**>> (p. 199) *Specifies how a writer should handle previously written samples to a new reader.*

6.60.1 Detailed Description

Specifies the behavior of RTI Connext in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers.

This QoS policy specifies how much data must to stored by RTI Connext for a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087). It controls whether RTI Connext should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between.

On the publishing side, this QoS policy controls the samples that should be maintained by the **DDSDataWriter** (p. 1113) on behalf of existing **DDSDataReader** (p. 1087) entities. The behavior with regards to a **DDSDataReader** (p. 1087) entities discovered after a sample is written is controlled by the **DURABILITY** (p. 348) policy.

On the subscribing side, this QoS policy controls the samples that should be maintained until the application "takes" them from RTI Connext.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

See also:

DDS_ReliabilityQosPolicy (p. 865)
DDS_HistoryQosPolicy (p. 758)

6.60.2 Usage

This policy controls the behavior of RTI Connext when the value of an instance changes before it is finally communicated to **DDSDataReader** (p. 1087) entities.

When a **DDSDataWriter** (p. 1113) sends data, or a **DDSDataReader** (p. 1087) receives data, the data sent or received is stored in a cache whose contents are controlled by this QoS policy. This QoS policy interacts with **DDS_ReliabilityQosPolicy** (p. 865) by controlling whether RTI Connext guarantees that *all* of the sent data is received (**DDS_KEEP_ALL_HISTORY_QOS** (p. 368)) or if only the last N data values sent are guaranteed to be received (**DDS_KEEP_ALL_HISTORY_QOS** (p. 368))–this is a reduced level of reliability.

The amount of data that is sent to new DataReaders who have configured their **DDS_DurabilityQosPolicy** (p. 614) to receive previously published data is also controlled by the History QoS policy.

Note that the History QoS policy does not control the *physical* sizes of the send and receive queues. The memory allocation for the queues is controlled by the **DDS_ResourceLimitsQosPolicy** (p. 879).

If `kind` is **DDS_KEEP_LAST_HISTORY_QOS** (p. 368) (the default), then RTI Connext will only attempt to keep the latest values of the instance and discard the older ones. In this case, the value of `depth` regulates the maximum number of values (up to and including the most current one) RTI Connext will maintain and deliver. After N values have been sent or received, any new data will overwrite the oldest data in the queue. Thus the queue acts like a circular buffer of length N.

The default (and most common setting) for `depth` is 1, indicating that only the most recent value should be delivered.

If `kind` is **DDS_KEEP_ALL_HISTORY_QOS** (p. 368), then RTI Connext will attempt to maintain and deliver all the values of the instance to existing subscribers. The resources that RTI Connext can use to keep this history are limited by the settings of the **RESOURCE_LIMITS** (p. 371). If the limit is reached, then the behavior of RTI Connext will depend on the **RELIABILITY**

(p. 362). If the Reliability `kind` is **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363), then the old values will be discarded. If Reliability `kind` is RELIABLE, then RTI Connext will block the **DDSDataWriter** (p. 1113) until it can deliver the necessary old values to all subscribers.

If `refilter` is **DDS_NONE_REFILTER_QOS** (p. 368), then samples written before a DataReader is matched to a DataWriter are not refiltered by the DataWriter.

If `refilter` is **DDS_ALL_REFILTER_QOS** (p. 369), then all samples written before a DataReader is matched to a DataWriter are refiltered by the DataWriter when the DataReader is matched.

If `refilter` is **DDS_ON_DEMAND_REFILTER_QOS** (p. 369), then a DataWriter will only refilter samples that a DataReader requests.

6.60.3 Consistency

This QoS policy's `depth` must be consistent with the **RESOURCE_LIMITS** (p. 371) `max_samples_per_instance`. For these two QoS to be consistent, they must verify that *depth* <= *max_samples_per_instance*.

See also:

DDS_ResourceLimitsQosPolicy (p. 879)

6.60.4 Member Data Documentation

6.60.4.1 DDS_HistoryQosPolicyKind DDS_HistoryQosPolicy::kind

Specifies the kind of history to be kept.

[default] DDS_KEEP_LAST_HISTORY_QOS (p. 368)

6.60.4.2 DDS_Long DDS_HistoryQosPolicy::depth

Specifies the number of samples to be kept, when the `kind` is **DDS_KEEP_LAST_HISTORY_QOS** (p. 368).

If a value other than 1 (the default) is specified, it should be consistent with the settings of the **RESOURCE_LIMITS** (p. 371) policy. That is:

depth <= **DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882)

When the `kind` is **DDS_KEEP_ALL_HISTORY_QOS** (p. 368), the depth has no effect. Its implied value is `infinity` (in practice limited by the settings of the **RESOURCE_LIMITS** (p. 371) policy).

[**default**] 1

[**range**] [1,100 million], <= **DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882)

6.60.4.3 DDS_RefilterQosPolicyKind DDS_HistoryQosPolicy::refilter

<<***eXtension***>> *(p. 199)* Specifies how a writer should handle previously written samples to a new reader.

[**default**] **DDS_NONE_REFILTER_QOS** (p. 368)

6.61 DDS_InconsistentTopicStatus Struct Reference

DDS_INCONSISTENT_TOPIC_STATUS (p. 322)

Public Attributes

- **DDS_Long total_count**

  *Total cumulative count of the Topics discovered whose name matches the **DDSTopic** (p. 1419) to which this status is attached and whose type is inconsistent with that of that **DDSTopic** (p. 1419).*

- **DDS_Long total_count_change**

  The incremental number of inconsistent topics discovered since the last time this status was read.

6.61.1 Detailed Description

DDS_INCONSISTENT_TOPIC_STATUS (p. 322)

Entity:

DDSTopic (p. 1419)

Listener:

DDSTopicListener (p. 1430)

A remote **DDSTopic** (p. 1419) will be inconsistent with the locally created **DDSTopic** (p. 1419) if the type name of the two topics are different.

6.61.2 Member Data Documentation

6.61.2.1 DDS_Long DDS_InconsistentTopicStatus::total_count

Total cumulative count of the Topics discovered whose name matches the **DDSTopic** (p. 1419) to which this status is attached and whose type is inconsistent with that of that **DDSTopic** (p. 1419).

6.61.2.2 DDS_Long DDS_InconsistentTopicStatus::total_count_change

The incremental number of inconsistent topics discovered since the last time this status was read.

6.62 DDS_InstanceHandleSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_InstanceHandle_t** (p. 53) > .

6.62.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_InstanceHandle_t** (p. 53) > .

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_InstanceHandle_t (p. 53)
FooSeq (p. 1494)

6.63 DDS_KeyedOctets Struct Reference

Built-in type consisting of a variable-length array of opaque bytes and a string that is the key.

Public Member Functions

- **DDS_KeyedOctets** ()
  Constructor.
- **DDS_KeyedOctets** (int key_size, int size)
  Constructor that specifies the allocated sizes.
- ~**DDS_KeyedOctets** ()
  Destructor.

Public Attributes

- char * **key**
  Instance key associated with the specified value.
- int **length**
  Number of octets to serialize.
- unsigned char * **value**
  DDS_Octets *(p. 799) array value.*

6.63.1 Detailed Description

Built-in type consisting of a variable-length array of opaque bytes and a string that is the key.

6.63.2 Constructor & Destructor Documentation

6.63.2.1 DDS_KeyedOctets::DDS_KeyedOctets () [inline]

Constructor.

The default constructor initializes the newly created object with NULL key, NULL value, and zero length.

6.63.2.2 DDS_KeyedOctets::DDS_KeyedOctets (int *key_size*, int *size*) [inline]

Constructor that specifies the allocated sizes.

After this method is called, key is initialized with the empty string and length is set to zero.

If a memory allocation failure occurs, and the **DDS_KeyedOctets** (p. 765) structure is allocated but the array and/or key string inside of it cannot be, the unallocated member will be NULL.

Parameters:

key_size <<*in*>> *(p. 200)* Size of the allocated key (with NULL-terminated character).

size <<*in*>> *(p. 200)* Size of the allocated octets array.

6.63.2.3 DDS_KeyedOctets::~DDS_KeyedOctets () [inline]

Destructor.

6.63.3 Member Data Documentation

6.63.3.1 char* DDS_KeyedOctets::key

Instance key associated with the specified value.

6.63.3.2 int DDS_KeyedOctets::length

Number of octets to serialize.

6.63.3.3 unsigned char* DDS_KeyedOctets::value

DDS_Octets (p. 799) array value.

6.64 DDS_KeyedOctetsSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_KeyedOctets** (p. 765) >.

6.64.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_KeyedOctets** (p. 765) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_KeyedOctets (p. 765)

6.65 DDS_KeyedString Struct Reference

Keyed string built-in type.

Public Member Functions

- **DDS_KeyedString** ()

  Constructor.

- **DDS_KeyedString** (int key_size, int size)

  Constructor that specifies the allocated sizes.

- ~**DDS_KeyedString** ()

  Destructor.

Public Attributes

- char * **key**

  Instance key associated with the specified value.

- char * **value**

  String value.

6.65.1 Detailed Description

Keyed string built-in type.

6.65.2 Constructor & Destructor Documentation

6.65.2.1 DDS_KeyedString::DDS_KeyedString () [inline]

Constructor.

The default constructor initializes the newly created object with NULL key and value.

6.65.2.2 DDS_KeyedString::DDS_KeyedString (int *key_size*, int *size*) [inline]

Constructor that specifies the allocated sizes.

The allocated strings are initialized to empty ("").

If a memory allocation failure occurs, and the **DDS_KeyedString** (p. 768) structure is allocated but one or both strings inside of it cannot be, the unallocated string will be NULL.

Parameters:

key_size <<*in*>> *(p. 200)* Size of the allocated key string (with NULL-terminated character).

size <<*in*>> *(p. 200)* Size of the allocated value string (with NULL-terminated character).

Returns:

A new **DDS_KeyedString** (p. 768) or NULL if failure.

6.65.2.3 DDS_KeyedString::~DDS_KeyedString () [inline]

Destructor.

6.65.3 Member Data Documentation

6.65.3.1 char* DDS_KeyedString::key

Instance key associated with the specified value.

6.65.3.2 char* DDS_KeyedString::value

String value.

6.66 DDS_KeyedStringSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_KeyedString** (p. 768) > .

6.66.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_KeyedString** (p. 768) > .

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_KeyedString (p. 768)

6.67 DDS_LatencyBudgetQosPolicy Struct Reference

Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

Public Attributes

- struct **DDS_Duration_t duration**
  Duration of the maximum acceptable delay.

6.67.1 Detailed Description

Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

This policy is a *hint* to a DDS implementation; it can be used to change how it processes and sends data that has low latency requirements. The DDS specification does not mandate whether or how this policy is used.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **YES** (p. 340)

See also:

DDS_PublishModeQosPolicy (p. 853)
DDSFlowController (p. 1259)

6.67.2 Usage

This policy provides a means for the application to indicate to the middleware the urgency of the data communication. By having a non-zero `duration`, RTI Connext can optimize its internal operation.

RTI Connext uses it in conjunction with **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) **DDSDataWriter** (p. 1113) instances associated with a **DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90) **DDSFlowController** (p. 1259) only. Together with the time of write, **DDS_LatencyBudgetQosPolicy::duration** (p. 772) determines the deadline of each individual sample. RTI Connext uses this information to prioritize the sending of asynchronously published data; see **DDS_AsynchronousPublisherQosPolicy** (p. 466).

6.67.3 Compatibility

The value offered is considered compatible with the value requested if and only if the inequality *offered duration* <= *requested duration* evaluates to 'TRUE'.

6.67.4 Member Data Documentation

6.67.4.1 struct DDS_Duration_t DDS_LatencyBudgetQosPolicy::duration [read]

Duration of the maximum acceptable delay.

[default] 0 (meaning minimize the delay)

6.68 DDS_LifespanQosPolicy Struct Reference

Specifies how long the data written by the **DDSDataWriter** (p. 1113) is considered valid.

Public Attributes

- struct **DDS_Duration_t duration**
  Maximum duration for the data's validity.

6.68.1 Detailed Description

Specifies how long the data written by the **DDSDataWriter** (p. 1113) is considered valid.

Each data sample written by the **DDSDataWriter** (p. 1113) has an associated expiration time beyond which the data should not be delivered to any application. Once the sample expires, the data will be removed from the **DDSDataReader** (p. 1087) caches as well as from the transient and persistent information caches.

The expiration time of each sample from the **DDSDataWriter** (p. 1113)'s cache is computed by adding the duration specified by this QoS policy to the sample's source timestamp. The expiration time of each sample from the **DDSDataReader** (p. 1087)'s cache is computed by adding the duration to the reception timestamp.

See also:

FooDataWriter::write (p. 1484)
FooDataWriter::write_w_timestamp (p. 1486)

Entity:

DDSTopic (p. 1419), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **YES** (p. 340)

6.68.2 Usage

The Lifespan QoS policy can be used to control how much data is stored by RTI Connext. Even if it is configured to store "all" of the data sent or received

for a topic (see **DDS_HistoryQosPolicy** (p. 758)), the total amount of data it stores may be limited by this QoS policy.

You may also use this QoS policy to ensure that applications do not receive or act on data, commands or messages that are too old and have 'expired.'

To avoid inconsistencies, multiple writers of the same instance should have the same lifespan.

See also:

DDS_SampleInfo::source_timestamp (p. 917)
DDS_SampleInfo::reception_timestamp (p. 920)

6.68.3 Member Data Documentation

6.68.3.1 struct DDS_Duration_t DDS_LifespanQosPolicy::duration [read]

Maximum duration for the data's validity.

[default] DDS_DURATION_INFINITE (p. 305)

[range] [1 nanosec, 1 year] or **DDS_DURATION_INFINITE** (p. 305)

6.69 DDS_LivelinessChangedStatus Struct Reference

DDS_LIVELINESS_CHANGED_STATUS (p. 325)

Public Attributes

- **DDS_Long alive_count**

  The total count of currently alive ***DDSDataWriter*** *(p. 1113) entities that write the* ***DDSTopic*** *(p. 1419) the* ***DDSDataReader*** *(p. 1087) reads.*

- **DDS_Long not_alive_count**

  The total count of currently not_alive ***DDSDataWriter*** *(p. 1113) entities that write the* ***DDSTopic*** *(p. 1419) the* ***DDSDataReader*** *(p. 1087) reads.*

- **DDS_Long alive_count_change**

  The change in the alive_count since the last time the listener was called or the status was read.

- **DDS_Long not_alive_count_change**

  The change in the not_alive_count since the last time the listener was called or the status was read.

- **DDS_InstanceHandle_t last_publication_handle**

  An instance handle to the last remote writer to change its liveliness.

6.69.1 Detailed Description

DDS_LIVELINESS_CHANGED_STATUS (p. 325)

The **DDSDataReaderListener::on_liveliness_changed** (p. 1110) callback may be invoked for the following reasons:

- Liveliness is truly lost - a sample has not been received within the time-frame specified in **DDS_LivelinessQosPolicy::lease_duration** (p. 782)
- Liveliness is recovered after being lost.
- A new matching entity has been discovered.
- A QoS has changed such that a pair of matching entities are no longer matching (such as a change to the **DDS_PartitionQosPolicy** (p. 820)). In this case, RTI Connext will no longer keep track of the entities' liveliness. Furthermore:

- If liveliness was maintained: **DDS_LivelinessChangedStatus::alive_count** (p. 776) will decrease and **DDS_LivelinessChangedStatus::not_alive_count** (p. 776) will remain the same.
- If liveliness had been lost: **DDS_LivelinessChangedStatus::alive_count** (p. 776) will remain the same and **DDS_LivelinessChangedStatus::not_alive_count** (p. 776) will decrease.

Examples:

HelloWorld_subscriber.cxx.

6.69.2 Member Data Documentation

6.69.2.1 DDS_Long DDS_LivelinessChangedStatus::alive_count

The total count of currently alive **DDSDataWriter** (p. 1113) entities that write the **DDSTopic** (p. 1419) the **DDSDataReader** (p. 1087) reads.

6.69.2.2 DDS_Long DDS_LivelinessChangedStatus::not_alive_count

The total count of currently not_alive **DDSDataWriter** (p. 1113) entities that write the **DDSTopic** (p. 1419) the **DDSDataReader** (p. 1087) reads.

6.69.2.3 DDS_Long DDS_LivelinessChangedStatus::alive_count_change

The change in the alive_count since the last time the listener was called or the status was read.

6.69.2.4 DDS_Long DDS_LivelinessChangedStatus::not_alive_count_change

The change in the not_alive_count since the last time the listener was called or the status was read.

6.69.2.5 DDS_InstanceHandle_t DDS_LivelinessChangedStatus::last_publication_handle

An instance handle to the last remote writer to change its liveliness.

6.70 DDS_LivelinessLostStatus Struct Reference

DDS_LIVELINESS_LOST_STATUS (p. 325)

Public Attributes

- **DDS_Long total_count**

  *Total cumulative number of times that a previously-alive **DDSDataWriter** (p. 1113) became not alive due to a failure to to actively signal its liveliness within the offered liveliness period.*

- **DDS_Long total_count_change**

  The incremental changees in total_count since the last time the listener was called or the status was read.

6.70.1 Detailed Description

DDS_LIVELINESS_LOST_STATUS (p. 325)

Entity:

DDSDataWriter (p. 1113)

Listener:

DDSDataWriterListener (p. 1133)

The liveliness that the **DDSDataWriter** (p. 1113) has committed through its **DDS_LivelinessQosPolicy** (p. 779) was not respected; thus **DDSDataReader** (p. 1087) entities will consider the **DDSDataWriter** (p. 1113) as no longer "alive/active".

6.70.2 Member Data Documentation

6.70.2.1 DDS_Long DDS_LivelinessLostStatus::total_count

Total cumulative number of times that a previously-alive **DDSDataWriter** (p. 1113) became not alive due to a failure to to actively signal its liveliness within the offered liveliness period.

This count does not change when an already not alive **DDSDataWriter** (p. 1113) simply remains not alive for another liveliness period.

6.70.2.2 DDS_Long DDS_LivelinessLostStatus::total_count_change

The incremental changees in total_count since the last time the listener was called or the status was read.

6.71 DDS_LivelinessQosPolicy Struct Reference

Specifies and configures the mechanism that allows **DDSDataReader** (p. 1087) entities to detect when **DDSDataWriter** (p. 1113) entities become disconnected or "dead.".

Public Attributes

- **DDS_LivelinessQosPolicyKind kind**

  The kind of liveliness desired.

- struct **DDS_Duration_t lease_duration**

  *The duration within which a **DDSEntity** (p. 1253) must be asserted, or else it is assumed to be not alive.*

6.71.1 Detailed Description

Specifies and configures the mechanism that allows **DDSDataReader** (p. 1087) entities to detect when **DDSDataWriter** (p. 1113) entities become disconnected or "dead.".

Liveliness must be asserted at least once every `lease_duration` otherwise RTI Connext will assume the corresponding **DDSEntity** (p. 1253) or is no longer alive.

The liveliness status of a **DDSEntity** (p. 1253) is used to maintain instance ownership in combination with the setting of the **OWNERSHIP** (p. 355) policy. The application is also informed via **DDSListener** (p. 1318) when an **DDSEntity** (p. 1253) is no longer alive.

A **DDSDataReader** (p. 1087) requests that liveliness of writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the `lease_duration`.

A **DDSDataWriter** (p. 1113) commits to signalling its liveliness using the stated means at intervals not to exceed the `lease_duration`.

Listeners are used to notify a **DDSDataReader** (p. 1087) of loss of liveliness and **DDSDataWriter** (p. 1113) of violations to the liveliness contract. The on_liveliness_lost() callback is only called *once*, after the first time the `lease_duration` is exceeded (when the **DDSDataWriter** (p. 1113) first loses liveliness).

This QoS policy can be used during system integration to ensure that applications have been coded to meet design specifications. It can also be used during run time to detect when systems are performing outside of design specifications.

Receiving applications can take appropriate actions in response to disconnected DataWriters.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_LIVELINESS_LOST_STATUS (p. 325), **DDS_LivelinessLostStatus** (p. 777);
DDS_LIVELINESS_CHANGED_STATUS (p. 325), **DDS_LivelinessChangedStatus** (p. 775);
DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.71.2 Usage

This policy controls the mechanism and parameters used by RTI Connext to ensure that particular entities on the network are still alive. The liveliness can also affect the ownership of a particular instance, as determined by the **OWNERSHIP** (p. 355) policy.

This policy has several settings to support both data types that are updated periodically as well as those that are changed sporadically. It also allows customisation for different application requirements in terms of the kinds of failures that will be detected by the liveliness mechanism.

The **DDS_AUTOMATIC_LIVELINESS_QOS** (p. 359) liveliness setting is most appropriate for applications that only need to detect failures at the process-level, but not application-logic failures within a process. RTI Connext takes responsibility for renewing the leases at the required rates and thus, as long as the local process where a **DDSDomainParticipant** (p. 1139) is running and the link connecting it to remote participants remains connected, the entities within the **DDSDomainParticipant** (p. 1139) will be considered alive. This requires the lowest overhead.

The manual settings (**DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** (p. 359), **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS** (p. 359)) require the application on the publishing side to periodically assert the liveliness before the lease expires to indicate the corresponding **DDSEntity** (p. 1253) is still alive. The action can be explicit

by calling the **DDSDataWriter::assert_liveliness** (p. 1121) operation or implicit by writing some data.

The two possible manual settings control the granularity at which the application must assert liveliness.

- The setting **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** (p. 359) requires only that one **DDSEntity** (p. 1253) within a participant is asserted to be alive to deduce all other **DDSEntity** (p. 1253) objects within the same **DDSDomainParticipant** (p. 1139) are also alive.

- The setting **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS** (p. 359) requires that at least one instance within the **DDSDataWriter** (p. 1113) is asserted.

Changes in **LIVELINESS** (p. 358) must be detected by the Service with a time-granularity greater or equal to the `lease_duration`. This ensures that the value of the **DDS_LivelinessChangedStatus** (p. 775) is updated at least once during each `lease_duration` and the related Listeners and **DDSWaitSet** (p. 1433) s are notified within a `lease_duration` from the time the **LIVELINESS** (p. 358) changed.

6.71.3 Compatibility

The value offered is considered compatible with the value requested if and only if the following conditions are met:

- the inequality *offered kind* >= *requested kind* evaluates to 'TRUE'. For the purposes of this inequality, the values of **DDS_LivelinessQosPolicyKind** (p. 358) kind are considered ordered such that: **DDS_AUTOMATIC_LIVELINESS_QOS** (p. 359) < **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** (p. 359) < **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS** (p. 359).

- the inequality *offered* `lease_duration` <= *requested* `lease_duration` evaluates to **DDS_BOOLEAN_TRUE** (p. 298).

See also:

RELATIONSHIP BETWEEN REGISTRATION, LIVELINESS and OWNERSHIP (p. 809)

6.71.4 Member Data Documentation

6.71.4.1 DDS_LivelinessQosPolicyKind DDS_LivelinessQosPolicy::kind

The kind of liveliness desired.

[default] **DDS_AUTOMATIC_LIVELINESS_QOS** (p. 359)

6.71.4.2 struct DDS_Duration_t DDS_LivelinessQosPolicy::lease_duration [read]

The duration within which a **DDSEntity** (p. 1253) must be asserted, or else it is assumed to be not alive.

[default] **DDS_DURATION_INFINITE** (p. 305)

[range] [0,1 year] or **DDS_DURATION_INFINITE** (p. 305)

6.72 DDS_Locator_t Struct Reference

<<*eXtension*>> *(p. 199)* Type used to represent the addressing information needed to send a message to an RTPS Endpoint using one of the supported transports.

Public Attributes

- **DDS_Long kind**

  The kind of locator.

- **DDS_UnsignedLong port**

  the port number

- **DDS_Octet address** [DDS_LOCATOR_ADDRESS_LENGTH_MAX]

  *A **DDS_LOCATOR_ADDRESS_LENGTH_MAX** (p. 45) octet field to hold the IP address.*

6.72.1 Detailed Description

<<*eXtension*>> *(p. 199)* Type used to represent the addressing information needed to send a message to an RTPS Endpoint using one of the supported transports.

6.72.2 Member Data Documentation

6.72.2.1 DDS_Long DDS_Locator_t::kind

The kind of locator.

If the Locator_t kind is **DDS_LOCATOR_KIND_UDPv4** (p. 47), the address contains an IPv4 address. In this case, the leading 12 octets of the **DDS_Locator_t::address** (p. 784) must be zero. The last 4 octets of **DDS_Locator_t::address** (p. 784) are used to store the IPv4 address.

If the Locator_t kind is **DDS_LOCATOR_KIND_UDPv6** (p. 47), the address contains an IPv6 address. IPv6 addresses typically use a shorthand hexadecimal notation that maps one-to-one to the 16 octets in the **DDS_Locator_t::address** (p. 784) field.

6.72.2.2 DDS_UnsignedLong DDS_Locator_t::port

the port number

6.72.2.3 DDS_Octet DDS_Locator_t::address[DDS_LOCATOR_ADDRESS_LENGTH_MAX]

A **DDS_LOCATOR_ADDRESS_LENGTH_MAX** (p. 45) octet field to hold the IP address.

6.73 DDS_LocatorFilter_t Struct Reference

Specifies the configuration of an individual channel within a MultiChannel DataWriter.

Public Attributes

- struct **DDS_LocatorSeq locators**

  *Sequence containing from one to four **DDS_Locator_t** (p. 783), used to specify the multicast address locators of an individual channel within a MultiChannel DataWriter.*

- char * **filter_expression**

  A logical expression used to determine the data that will be published in the channel.

6.73.1 Detailed Description

Specifies the configuration of an individual channel within a MultiChannel DataWriter.

QoS:

DDS_LocatorFilterQosPolicy (p. 787)

6.73.2 Member Data Documentation

6.73.2.1 struct DDS_LocatorSeq DDS_LocatorFilter_t::locators [read]

Sequence containing from one to four **DDS_Locator_t** (p. 783), used to specify the multicast address locators of an individual channel within a MultiChannel DataWriter.

[**default**] Empty sequence.

6.73.2.2 char* DDS_LocatorFilter_t::filter_expression

A logical expression used to determine the data that will be published in the channel.

If the expression evaluates to TRUE, a sample will be published on the channel.

An empty string always evaluates the expression to TRUE.

A NULL value is not allowed.

The syntax of the expression will depend on the value of **DDS_LocatorFilterQosPolicy::filter_name** (p. 788)

Important: This value must be an allocated string with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459). It should not be assigned to a string constant.

See also:

Queries and Filters Syntax (p. 208)

[default] NULL (invalid value)

6.74 DDS_LocatorFilterQosPolicy Struct Reference

The QoS policy used to report the configuration of a MultiChannel DataWriter as part of **DDS_PublicationBuiltinTopicData** (p. 839).

Public Attributes

- struct **DDS_LocatorFilterSeq locator_filters**

  *A sequence of **DDS_LocatorFilter_t** (p. 785). Each **DDS_LocatorFilter_t** (p. 785) reports the configuration of a single channel of a MultiChannel DataWriter.*

- char * **filter_name**

  Name of the filter class used to describe the filter expressions of a MultiChannel DataWriter.

6.74.1 Detailed Description

The QoS policy used to report the configuration of a MultiChannel DataWriter as part of **DDS_PublicationBuiltinTopicData** (p. 839).

Entity:

DDS_PublicationBuiltinTopicData (p. 839)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.74.2 Member Data Documentation

6.74.2.1 struct DDS_LocatorFilterSeq DDS_LocatorFilterQosPolicy::locator_filters [read]

A sequence of **DDS_LocatorFilter_t** (p. 785). Each **DDS_LocatorFilter_t** (p. 785) reports the configuration of a single channel of a MultiChannel DataWriter.

A sequence length of zero indicates the **DDS_MultiChannelQosPolicy** (p. 796) is not in use.

[default] Empty sequence.

6.74.2.2 char* DDS_LocatorFilterQosPolicy::filter_name

Name of the filter class used to describe the filter expressions of a MultiChannel DataWriter.

The following builtin filters are supported: **DDS_SQLFILTER_NAME** (p. 41) and **DDS_STRINGMATCHFILTER_NAME** (p. 41).

Important: This value must be assigned to either one of the predefined values (**DDS_SQLFILTER_NAME** (p. 41) or **DDS_STRINGMATCHFILTER_NAME** (p. 41)), or to an allocated string with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459). It should not be assigned to a string constant.

[default] **DDS_STRINGMATCHFILTER_NAME** (p. 41)

6.75 DDS_LocatorFilterSeq Struct Reference

Declares IDL sequence< **DDS_LocatorFilter_t** (p. 785) >.

6.75.1 Detailed Description

Declares IDL sequence< **DDS_LocatorFilter_t** (p. 785) >.

A sequence of **DDS_LocatorFilter_t** (p. 785) used to report the channels' properties. If the length of the sequence is zero, the **DDS_MultiChannelQosPolicy** (p. 796) is not in use.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_LocatorFilter_t (p. 785)

6.76 DDS_LocatorSeq Struct Reference

Declares IDL `sequence` < **DDS_Locator_t** (p. 783) >.

6.76.1 Detailed Description

Declares IDL `sequence` < **DDS_Locator_t** (p. 783) >.

See also:

DDS_Locator_t (p. 783)

6.77 DDS_LoggingQosPolicy Struct Reference

Configures the RTI Connext logging facility.

Public Attributes

- **NDDS_Config_LogVerbosity verbosity**
  The verbosities at which RTI Connext diagnostic information is logged.
- **NDDS_Config_LogCategory category**
  Categories of logged messages.
- **NDDS_Config_LogPrintFormat print_format**
  The format used to output RTI Connext diagnostic information.
- char * **output_file**
  Specifies the file to which log messages will be redirected to.

6.77.1 Detailed Description

Configures the RTI Connext logging facility.

All the properties associated with RTI Connext logging can be configured using this QoS policy. This allows you to configure logging using XML QoS Profiles. See the Troubleshooting chapter in the `User's Manual` for details.

Entity:

DDSDomainParticipantFactory (p. 1216)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **Changeable** (p. 340)

6.77.2 Member Data Documentation

6.77.2.1 NDDS_Config_LogVerbosity DDS_LoggingQosPolicy::verbosity

The verbosities at which RTI Connext diagnostic information is logged.

[**default**] DDS_NDDS_CONFIG_LOG_VERBOSITY_ERROR

6.77.2.2 NDDS_Config_LogCategory DDS_LoggingQosPolicy::category

Categories of logged messages.

[default] Logging will be enabled for all the categories.

6.77.2.3 NDDS_Config_LogPrintFormat DDS_LoggingQosPolicy::print_format

The format used to output RTI Connext diagnostic information.

[default] DDS_NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT.

6.77.2.4 char* DDS_LoggingQosPolicy::output_file

Specifies the file to which log messages will be redirected to.

If the value of ouput_file is set to NULL, log messages will sent to standard output.

Important: This value must be an allocated string with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459). It should not be assigned to a string constant.

[default] NULL

6.78 DDS_LongDoubleSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_LongDouble** (p. 300) >.

6.78.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_LongDouble** (p. 300) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_LongDouble (p. 300)
FooSeq (p. 1494)

6.79 DDS_LongLongSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_LongLong** (p. 300) >.

6.79.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_LongLong** (p. 300) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_LongLong (p. 300)
FooSeq (p. 1494)

6.80 DDS_LongSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_Long** (p. 300) >.

6.80.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_Long** (p. 300) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Long (p. 300)
FooSeq (p. 1494)

6.81 DDS_MultiChannelQosPolicy Struct Reference

Configures the ability of a DataWriter to send data on different multicast groups (addresses) based on the value of the data.

Public Attributes

- struct **DDS_ChannelSettingsSeq channels**

  *A sequence of **DDS_ChannelSettings_t** (p. 486) used to configure the channels' properties. If the length of the sequence is zero, the QoS policy will be ignored.*

- char * **filter_name**

  Name of the filter class used to describe the filter expressions of a MultiChannel DataWriter.

6.81.1 Detailed Description

Configures the ability of a DataWriter to send data on different multicast groups (addresses) based on the value of the data.

This QoS policy is used to partition the data published by a **DDSDataWriter** (p. 1113) across multiple channels. A *channel* is defined by a filter expression and a sequence of multicast locators.

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.81.2 Usage

By using this QoS, a **DDSDataWriter** (p. 1113) can be configured to send data to different multicast groups based on the content of the data. Using syntax similar to those used in Content-Based Filters, you can associate different multicast addresses with filter expressions that operate on the values of the fields within the data. When your application's code calls **FooDataWriter::write**

(p. 1484), data is sent to any multicast address for which the data passes the filter.

Multi-channel DataWriters can be used to trade off network bandwidth with the unnecessary processing of unwanted data for situations where there are multiple DataReaders that are interested in different subsets of data that come from the same data stream (Topic). For example, in Financial applications, the data stream may be quotes for different stocks at an exchange. Applications usually only want to receive data (quotes) for only a subset of the stocks being traded. In tracking applications, a data stream may carry information on hundreds or thousands of objects being tracked, but again, applications may only be interested in a subset.

The problem is that the most efficient way to deliver data to multiple applications is to use multicast, so that a data value is only sent once on the network for any number of subscribers to the data. However, using multicast, an application will receive *all* of the data sent and not just the data in which it is interested, thus extra CPU time is wasted to throw away unwanted data. With this QoS, you can analyze the data-usage patterns of your applications and optimize network vs. CPU usage by partitioning the data into multiple multicast streams. While network bandwidth is still being conserved by sending data only once using multicast, most applications will only need to listen to a subset of the multicast addresses and receive a reduced amount of unwanted data.

Your system can gain more of the benefits of using multiple multicast groups if your network uses Layer 2 Ethernet switches. Layer 2 switches can be configured to only route multicast packets to those ports that have added membership to specific multicast groups. Using those switches will ensure that only the multicast packets used by applications on a node are routed to the node; all others are filtered-out by the switch.

6.81.3 Member Data Documentation

6.81.3.1 struct DDS_ChannelSettingsSeq DDS_MultiChannelQosPolicy::channels [read]

A sequence of **DDS_ChannelSettings_t** (p. 486) used to configure the channels' properties. If the length of the sequence is zero, the QoS policy will be ignored.

A sequence length of zero indicates the **DDS_MultiChannelQosPolicy** (p. 796) is not in use.

The sequence length cannot be greater than **DDS_DomainParticipantResourceLimitsQosPolicy::channel_seq_max_length** (p. 609).

[**default**] Empty sequence.

6.81.3.2 char* DDS_MultiChannelQosPolicy::filter_name

Name of the filter class used to describe the filter expressions of a MultiChannel DataWriter.

The following builtin filters are supported: **DDS_SQLFILTER_NAME** (p. 41) and **DDS_STRINGMATCHFILTER_NAME** (p. 41).

Important: This value must be assigned to either one of the predefined values (**DDS_SQLFILTER_NAME** (p. 41) or **DDS_STRINGMATCHFILTER_NAME** (p. 41)), or to an allocated string with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459). It should not be assigned to a string constant.

[default] DDS_STRINGMATCHFILTER_NAME (p. 41)

6.82 DDS_Octets Struct Reference

Built-in type consisting of a variable-length array of opaque bytes.

Public Member Functions

- **DDS_Octets** ()
  Constructor.
- **DDS_Octets** (int size)
  Constructor that specifies the size of the allocated octets array.
- **~DDS_Octets** ()
  Destructor.

Public Attributes

- int **length**
  Number of octets to serialize.
- unsigned char * **value**
  DDS_Octets *(p. 799) array value.*

6.82.1 Detailed Description

Built-in type consisting of a variable-length array of opaque bytes.

6.82.2 Constructor & Destructor Documentation

6.82.2.1 DDS_Octets::DDS_Octets () [inline]

Constructor.

The default constructor initializes the newly created object with NULL value, and zero length.

6.82.2.2 DDS_Octets::DDS_Octets (int *size*) [inline]

Constructor that specifies the size of the allocated octets array.

After this method is called, length is set to zero.

If a memory allocation failure occurs, and the **DDS_Octets** (p. 799) structure is allocated but the array inside of it cannot be, the array will be NULL.

Parameters:

size *<<**in**>>* *(p. 200)* Size of the allocated octets array.

6.82.2.3 DDS_Octets::~DDS_Octets () [inline]

Destructor.

6.82.3 Member Data Documentation

6.82.3.1 int DDS_Octets::length

Number of octets to serialize.

6.82.3.2 unsigned char* DDS_Octets::value

DDS_Octets (p. 799) array value.

6.83 DDS_OctetSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_Octet** (p. 299) >.

6.83.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_Octet** (p. 299) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Octet (p. 299)
FooSeq (p. 1494)

6.84 DDS_OctetsSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_Octets** (p. 799) > .

6.84.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_Octets** (p. 799) > .

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Octets (p. 799)

6.85 DDS_OfferedDeadlineMissedStatus Struct Reference

DDS_OFFERED_DEADLINE_MISSED_STATUS (p. 323)

Public Attributes

- **DDS_Long total_count**

  *Total cumulative count of the number of times the **DDSDataWriter** (p. 1113) failed to write within its offered deadline.*

- **DDS_Long total_count_change**

  The incremental changes in total_count since the last time the listener was called or the status was read.

- **DDS_InstanceHandle_t last_instance_handle**

  *Handle to the last instance in the **DDSDataWriter** (p. 1113) for which an offered deadline was missed.*

6.85.1 Detailed Description

DDS_OFFERED_DEADLINE_MISSED_STATUS (p. 323)

Entity:

DDSDataWriter (p. 1113)

Listener:

DDSDataWriterListener (p. 1133)

The deadline that the **DDSDataWriter** (p. 1113) has committed through its **DDS_DeadlineQosPolicy** (p. 567) was not respected for a specific instance.

6.85.2 Member Data Documentation

6.85.2.1 DDS_Long DDS_OfferedDeadlineMissedStatus::total_count

Total cumulative count of the number of times the **DDSDataWriter** (p. 1113) failed to write within its offered deadline.

Missed deadlines accumulate; that is, each deadline period the `total_count` will be incremented by one.

6.85.2.2 DDS_Long DDS_OfferedDeadlineMissedStatus::total_count_change

The incremental changes in total_count since the last time the listener was called or the status was read.

6.85.2.3 DDS_InstanceHandle_t DDS_OfferedDeadlineMissedStatus::last_instance_handle

Handle to the last instance in the **DDSDataWriter** (p. 1113) for which an offered deadline was missed.

6.86 DDS_OfferedIncompatibleQosStatus Struct Reference

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323)

Public Attributes

- **DDS_Long total_count**

  *Total cumulative number of times the concerned **DDSDataWriter** (p. 1113) discovered a **DDSDataReader** (p. 1087) for the same **DDSTopic** (p. 1419), common partition with a requested QoS that is incompatible with that offered by the **DDSDataWriter** (p. 1113).*

- **DDS_Long total_count_change**

  The incremental changes in total_count since the last time the listener was called or the status was read.

- **DDS_QosPolicyId_t last_policy_id**

  *The **DDS_QosPolicyId_t** (p. 341) of one of the policies that was found to be incompatible the last time an incompatibility was detected.*

- struct **DDS_QosPolicyCountSeq policies**

  *A list containing for each policy the total number of times that the concerned **DDSDataWriter** (p. 1113) discovered a **DDSDataReader** (p. 1087) for the same **DDSTopic** (p. 1419) and common partition with a requested QoS that is incompatible with that offered by the **DDSDataWriter** (p. 1113).*

6.86.1 Detailed Description

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323)

Entity:

DDSDataWriter (p. 1113)

Listener:

DDSDataWriterListener (p. 1133)

The qos policy value was incompatible with what was requested.

6.86.2 Member Data Documentation

6.86.2.1 DDS_Long DDS_OfferedIncompatibleQosStatus::total_count

Total cumulative number of times the concerned **DDSDataWriter** (p. 1113) discovered a **DDSDataReader** (p. 1087) for the same **DDSTopic** (p. 1419), common partition with a requested QoS that is incompatible with that offered by the **DDSDataWriter** (p. 1113).

6.86.2.2 DDS_Long DDS_OfferedIncompatibleQosStatus::total_count_change

The incremental changes in total_count since the last time the listener was called or the status was read.

6.86.2.3 DDS_QosPolicyId_t DDS_OfferedIncompatibleQosStatus::last_policy_id

The **DDS_QosPolicyId_t** (p. 341) of one of the policies that was found to be incompatible the last time an incompatibility was detected.

6.86.2.4 struct DDS_QosPolicyCountSeq DDS_OfferedIncompatibleQosStatus::policies [read]

A list containing for each policy the total number of times that the concerned **DDSDataWriter** (p. 1113) discovered a **DDSDataReader** (p. 1087) for the same **DDSTopic** (p. 1419) and common partition with a requested QoS that is incompatible with that offered by the **DDSDataWriter** (p. 1113).

6.87 DDS_OwnershipQosPolicy Struct Reference

Specifies whether it is allowed for multiple **DDSDataWriter** (p. 1113) (s) to write the same instance of the data and if so, how these modifications should be arbitrated.

Public Attributes

- **DDS_OwnershipQosPolicyKind kind**

  The kind of ownership.

6.87.1 Detailed Description

Specifies whether it is allowed for multiple **DDSDataWriter** (p. 1113) (s) to write the same instance of the data and if so, how these modifications should be arbitrated.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

See also:

OWNERSHIP_STRENGTH (p. 357)

6.87.2 Usage

Along with the **OWNERSHIP_STRENGTH** (p. 357), this QoS policy specifies if **DDSDataReader** (p. 1087) entities can receive updates to the same instance (identified by its key) from multiple **DDSDataWriter** (p. 1113) entities at the same time.

There are two kinds of ownership, selected by the setting of the `kind`: SHARED and EXCLUSIVE.

6.87.2.1 SHARED ownership

DDS_SHARED_OWNERSHIP_QOS (p. 356) indicates that RTI Connext does not enforce unique ownership for each instance. In this case, multiple writers can update the same data type instance. The subscriber to the **DDSTopic** (p. 1419) will be able to access modifications from all **DDSDataWriter** (p. 1113) objects, subject to the settings of other QoS that may filter particular samples (e.g. the **TIME_BASED_FILTER** (p. 360) or **HISTORY** (p. 367) policy). In any case, there is no "filtering" of modifications made based on the identity of the **DDSDataWriter** (p. 1113) that causes the modification.

6.87.2.2 EXCLUSIVE ownership

DDS_EXCLUSIVE_OWNERSHIP_QOS (p. 356) indicates that each instance of a data type can only be modified by one **DDSDataWriter** (p. 1113). In other words, at any point in time, a single **DDSDataWriter** (p. 1113) owns each instance and is the only one whose modifications will be visible to the **DDSDataReader** (p. 1087) objects. The owner is determined by selecting the **DDSDataWriter** (p. 1113) with the highest value of the **DDS_OwnershipStrengthQosPolicy::value** (p. 814) that is currently alive, as defined by the **LIVELINESS** (p. 358) policy, and has not violated its **DEADLINE** (p. 353) contract with regards to the data instance.

Ownership can therefore change as a result of:

- a **DDSDataWriter** (p. 1113) in the system with a higher value of the strength that modifies the instance,
- a change in the strength value of the **DDSDataWriter** (p. 1113) that owns the instance, and
- a change in the liveliness of the **DDSDataWriter** (p. 1113) that owns the instance.
- a deadline with regards to the instance that is missed by the **DDSDataWriter** (p. 1113) that owns the instance.

The behavior of the system is as if the determination was made independently by each **DDSDataReader** (p. 1087). Each **DDSDataReader** (p. 1087) may detect the change of ownership at a different time. It is not a requirement that at a particular point in time all the **DDSDataReader** (p. 1087) objects for that **DDSTopic** (p. 1419) have a consistent picture of who owns each instance.

It is also not a requirement that the **DDSDataWriter** (p. 1113) objects are aware of whether they own a particular instance. There is no error or notification given to a **DDSDataWriter** (p. 1113) that modifies an instance it does not currently own.

The requirements are chosen to (a) preserve the decoupling of publishers and subscriber, and (b) allow the policy to be implemented efficiently.

It is possible that multiple **DDSDataWriter** (p. 1113) objects with the same strength modify the same instance. If this occurs RTI Connext will pick one of the **DDSDataWriter** (p. 1113) objects as the owner. It is not specified how the owner is selected. However, the algorithm used to select the owner guarantees that all **DDSDataReader** (p. 1087) objects will make the same choice of the particular **DDSDataWriter** (p. 1113) that is the owner. It also guarantees that the owner remains the same until there is a change in strength, liveliness, the owner misses a deadline on the instance, or a new **DDSDataWriter** (p. 1113) with higher same strength, or a new **DDSDataWriter** (p. 1113) with same strength that should be deemed the owner according to the policy of the Service, modifies the instance.

Exclusive ownership is on an instance-by-instance basis. That is, a subscriber can receive values written by a lower strength **DDSDataWriter** (p. 1113) as long as they affect instances whose values have not been set by the higher-strength **DDSDataWriter** (p. 1113).

6.87.3 Compatibility

The value of the **DDS_OwnershipQosPolicyKind** (p. 355) offered must exactly match the one requested or else they are considered incompatible.

6.87.4 RELATIONSHIP BETWEEN REGISTRATION, LIVELINESS and OWNERSHIP

The need for registering/unregistering instances stems from two use cases:

- Ownership resolution on redundant systems
- Detection of loss in topological connectivity

These two use cases also illustrate the semantic differences between the **FooDataWriter::unregister_instance** (p. 1480) and **FooDataWriter::dispose** (p. 1489).

6.87.4.1 Ownership Resolution on Redundant Systems

It is expected that users may use DDS to set up redundant systems where multiple **DDSDataWriter** (p. 1113) entities are "capable" of writing the same instance. In this situation, the **DDSDataWriter** (p. 1113) entities are configured such that:

- Either both are writing the instance "constantly"
- Or else they use some mechanism to classify each other as "primary" and "secondary", such that the primary is the only one writing, and the secondary monitors the primary and only writes when it detects that the primary "writer" is no longer writing.

Both cases above use the **DDS_EXCLUSIVE_OWNERSHIP_QOS** (p. 356) and arbitrate themselves by means of the **DDS_OwnershipStrengthQosPolicy** (p. 814). Regardless of the scheme, the desired behavior from the **DDSDataReader** (p. 1087) point of view is that **DDSDataReader** (p. 1087) normally receives data from the primary unless the "primary" writer stops writing, in which case the **DDSDataReader** (p. 1087) starts to receive data from the secondary **DDSDataWriter** (p. 1113).

This approach requires some mechanism to detect that a **DDSDataWriter** (p. 1113) (the primary) is no longer "writing" the data as it should. There are several reasons why this may happen and all must be detected (but not necessarily distinguished):

crash The writing process is no longer running (e.g. the whole application has crashed)

connectivity loss Connectivity to the writing application has been lost (e.g. network disconnection)

application fault The application logic that was writing the data is faulty and has stopped calling **FooDataWriter::write** (p. 1484).

Arbitrating from a **DDSDataWriter** (p. 1113) to one of a higher strength is simple and the decision can be taken autonomously by the **DDSDataReader** (p. 1087). Switching ownership from a higher strength **DDSDataWriter** (p. 1113) to one of a lower strength **DDSDataWriter** (p. 1113) requires that the **DDSDataReader** (p. 1087) can make a determination that the stronger **DDSDataWriter** (p. 1113) is "no longer writing the instance".

Case where the data is periodically updated This determination is reasonably simple when the data is being written periodically at some rate. The **DDSDataWriter** (p. 1113) simply states its offered **DDS_DeadlineQosPolicy** (p. 567) (maximum interval between updates) and the **DDSDataReader** (p. 1087) automatically monitors that the **DDSDataWriter** (p. 1113) indeed updates the instance at least once per **DDS_DeadlineQosPolicy::period** (p. 569). If the deadline is missed, the **DDSDataReader** (p. 1087) considers the **DDSDataWriter** (p. 1113) "not alive" and automatically gives ownership to the next highest-strength **DDSDataWriter** (p. 1113) that *is* alive.

Case where data is not periodically updated The case where the **DDSDataWriter** (p. 1113) is not writing data periodically is also a very important use-case. Since the instance is not being updated at any fixed period, the "deadline" mechanism cannot be used to determine ownership. The liveliness solves this situation. Ownership is maintained while the **DDSDataWriter** (p. 1113) is "alive" and for the **DDSDataWriter** (p. 1113) to be alive it must fulfill its **DDS_LivelinessQosPolicy** (p. 779) contract. The different means to renew liveliness (automatic, manual) combined by the implied renewal each time data is written handle the three conditions above [crash], [connectivity loss], and [application fault]. Note that to handle [application fault], LIVELINESS must be **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS** (p. 359). The **DDSDataWriter** (p. 1113) can retain ownership by periodically writing data or else calling assert_liveliness if it has no data to write. Alternatively if only protection against [crash] or [connectivity loss] is desired, it is sufficient that some task on the **DDSDataWriter** (p. 1113) process periodically writes data or calls **DDSDomainParticipant::assert_liveliness** (p. 1192). However, this scenario requires that the **DDSDataReader** (p. 1087) knows what instances are being "written" by the **DDSDataWriter** (p. 1113). That is the only way that the **DDSDataReader** (p. 1087) deduces the ownership of specific instances from the fact that the **DDSDataWriter** (p. 1113) is still "alive". Hence the need for the **DDSDataWriter** (p. 1113) to "register" and "unregister" instances. Note that while "registration" can be done lazily the first time the **DDSDataWriter** (p. 1113) writes the instance, "unregistration," in general, cannot. Similar reasoning will lead to the fact that unregistration will also require a message to be sent to the **DDSDataReader** (p. 1087).

6.87.4.2 Detection of Loss in Topological Connectivity

There are applications that are designed in such a way that their correct operation requires some minimal topological connectivity, that is, the writer needs to have a minimum number of readers or alternatively the reader must have a minimum number of writers.

A common scenario is that the application does not start doing its logic until it

knows that some specific writers have the minimum configured readers (e.g the alarm monitor is up).

A *more* common scenario is that the application logic will wait until some writers appear that can provide some needed source of information (e.g. the raw sensor data that must be processed).

Furthermore, once the application is running it is a requirement that this minimal connectivity (from the source of the data) is monitored and the application informed if it is ever lost. For the case where data is being written periodically, the **DDS_DeadlineQosPolicy** (p. 567) and the on_deadline_missed listener provides the notification. The case where data is not periodically updated requires the use of the **DDS_LivelinessQosPolicy** (p. 779) in combination with register_instance/unregister_instance to detect whether the "connectivity" has been lost, and the notification is provided by means of **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117).

In terms of the required mechanisms, the scenario is very similar to the case of maintaining ownership. In both cases, the reader needs to know whether a writer is still "managing the current value of an instance" even though it is not continually writing it and this knowledge requires the writer to keep its liveliness plus some means to know which instances the writer is currently "managing" (i.e. the registered instances).

6.87.4.3 Semantic Difference between unregister_instance and dispose

FooDataWriter::dispose (p. 1489) is semantically different from **FooDataWriter::unregister_instance** (p. 1480). **FooDataWriter::dispose** (p. 1489) indicates that the data instance no longer exists (e.g. a track that has disappeared, a simulation entity that has been destroyed, a record entry that has been deleted, etc.) whereas **FooDataWriter::unregister_instance** (p. 1480) indicates that the writer is no longer taking responsibility for updating the value of the instance.

Deleting a **DDSDataWriter** (p. 1113) is equivalent to unregistering all the instances it was writing, but is *not* the same as "disposing" all the instances.

For a **DDSTopic** (p. 1419) with **DDS_EXCLUSIVE_OWNERSHIP_QOS** (p. 356), if the current owner of an instance *disposes* it, the readers accessing the instance will see the instance_state as being "DISPOSED" and not see the values being written by the weaker writer (even after the stronger one has disposed the instance). This is because the **DDSDataWriter** (p. 1113) that owns the instance is saying that the instance no longer exists (e.g. the master of the database is saying that a record has been deleted) and thus the readers should see it as such.

For a **DDSTopic** (p. 1419) with **DDS_EXCLUSIVE_OWNERSHIP_QOS**

(p. 356), if the current owner of an instance *unregisters* it, then it will relinquish ownership of the instance and thus the readers may see the value updated by another writer (which will then become the owner). This is because the owner said that it no longer will be providing values for the instance and thus another writer can take ownership and provide those values.

6.87.5 Member Data Documentation

6.87.5.1 DDS_OwnershipQosPolicyKind DDS_OwnershipQosPolicy::kind

The kind of ownership.

[**default**] **DDS_SHARED_OWNERSHIP_QOS** (p. 356)

6.88 DDS_OwnershipStrengthQosPolicy Struct Reference

Specifies the value of the strength used to arbitrate among multiple **DDSDataWriter** (p. 1113) objects that attempt to modify the same instance of a data type (identified by **DDSTopic** (p. 1419) + key).

Public Attributes

- **DDS_Long value**

  The strength value used to arbitrate among multiple writers.

6.88.1 Detailed Description

Specifies the value of the strength used to arbitrate among multiple **DDSDataWriter** (p. 1113) objects that attempt to modify the same instance of a data type (identified by **DDSTopic** (p. 1419) + key).

This policy only applies if the **OWNERSHIP** (p. 355) policy is of kind **DDS_EXCLUSIVE_OWNERSHIP_QOS** (p. 356).

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **YES** (p. 340)

The value of the **OWNERSHIP_STRENGTH** (p. 357) is used to determine the ownership of a data instance (identified by the key). The arbitration is performed by the **DDSDataReader** (p. 1087).

See also:

EXCLUSIVE ownership (p. 808)

6.88.2 Member Data Documentation

6.88.2.1 DDS_Long DDS_OwnershipStrengthQosPolicy::value

The strength value used to arbitrate among multiple writers.

[**default**] 0

[**range**] [0, 1 million]

6.89 DDS_ParticipantBuiltinTopicData Struct Reference

Entry created when a DomainParticipant object is discovered.

Public Attributes

- **DDS_BuiltinTopicKey_t key**

  DCPS key to distinguish entries.

- struct **DDS_UserDataQosPolicy user_data**

  Policy of the corresponding DomainParticipant.

- struct **DDS_PropertyQosPolicy property**

  <<**eXtension**>> (p. 199) *Name value pair properties to be stored with domain participant*

- **DDS_ProtocolVersion_t rtps_protocol_version**

  <<**eXtension**>> (p. 199) *Version number of the RTPS wire protocol used.*

- struct **DDS_VendorId_t rtps_vendor_id**

  <<**eXtension**>> (p. 199) *ID of vendor implementing the RTPS wire protocol.*

- **DDS_UnsignedLong dds_builtin_endpoints**

  <<**eXtension**>> (p. 199) *Bitmap of builtin endpoints supported by the participant.*

- struct **DDS_LocatorSeq default_unicast_locators**

  <<**eXtension**>> (p. 199) *Unicast locators used when individual entities do not specify unicast locators.*

- struct **DDS_ProductVersion_t product_version**

  <<**eXtension**>> (p. 199) *This is a vendor specific parameter. It gives the current version for rti-dds.*

- struct **DDS_EntityNameQosPolicy participant_name**

  <<**eXtension**>> (p. 199) *The participant name and role name.*

6.89.1 Detailed Description

Entry created when a DomainParticipant object is discovered.

Data associated with the built-in topic **DDS_PARTICIPANT_TOPIC_NAME** (p. 285). It contains QoS policies and additional information that apply to the remote **DDSDomainParticipant** (p. 1139).

See also:

DDS_PARTICIPANT_TOPIC_NAME (p. 285)
DDSParticipantBuiltinTopicDataDataReader (p. 1341)

6.89.2 Member Data Documentation

6.89.2.1 DDS_BuiltinTopicKey_t DDS_ParticipantBuiltinTopicData::key

DCPS key to distinguish entries.

6.89.2.2 struct DDS_UserDataQosPolicy DDS_ParticipantBuiltinTopicData::user_data [read]

Policy of the corresponding DomainParticipant.

6.89.2.3 struct DDS_PropertyQosPolicy DDS_ParticipantBuiltinTopicData::property [read]

<<***eXtension***>> *(p. 199)* Name value pair properties to be stored with domain participant

6.89.2.4 DDS_ProtocolVersion_t DDS_ParticipantBuiltinTopicData::rtps_protocol_version

<<***eXtension***>> *(p. 199)* Version number of the RTPS wire protocol used.

6.89.2.5 struct DDS_VendorId_t DDS_ParticipantBuiltinTopicData::rtps_vendor_id [read]

<<***eXtension***>> *(p. 199)* ID of vendor implementing the RTPS wire protocol.

6.89.2.6 DDS_UnsignedLong DDS_-ParticipantBuiltinTopicData::dds_builtin_endpoints

<<*eXtension*>> *(*p. *199)* Bitmap of builtin endpoints supported by the participant.

Each bit indicates a builtin endpoint that may be available on the participant for use in discovery.

6.89.2.7 struct DDS_LocatorSeq DDS_-ParticipantBuiltinTopicData::default_unicast_locators [read]

<<*eXtension*>> *(*p. *199)* Unicast locators used when individual entities do not specify unicast locators.

6.89.2.8 struct DDS_ProductVersion_t DDS_-ParticipantBuiltinTopicData::product_version [read]

<<*eXtension*>> *(*p. *199)* This is a vendor specific parameter. It gives the current version for rti-dds.

6.89.2.9 struct DDS_EntityNameQosPolicy DDS_-ParticipantBuiltinTopicData::participant_name [read]

<<*eXtension*>> *(*p. *199)* The participant name and role name.

This parameter contains the name and the role name of the discovered participant.

6.90 DDS_ParticipantBuiltinTopicDataSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_ParticipantBuiltinTopicData** (p. 816) > .

6.90.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_ParticipantBuiltinTopicData** (p. 816) > .

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_ParticipantBuiltinTopicData (p. 816)

6.91 DDS_PartitionQosPolicy Struct Reference

Set of strings that introduces a logical partition among the topics visible by a **DDSPublisher** (p. 1346) and a **DDSSubscriber** (p. 1390).

Public Attributes

- struct **DDS_StringSeq name**

  A list of partition names.

6.91.1 Detailed Description

Set of strings that introduces a logical partition among the topics visible by a **DDSPublisher** (p. 1346) and a **DDSSubscriber** (p. 1390).

This QoS policy is used to set string identifiers that are used for matching DataReaders and DataWriters for the same Topic.

A **DDSDataWriter** (p. 1113) within a **DDSPublisher** (p. 1346) only communicates with a **DDSDataReader** (p. 1087) in a **DDSSubscriber** (p. 1390) if (in addition to matching the **DDSTopic** (p. 1419) and having compatible QoS) the **DDSPublisher** (p. 1346) and **DDSSubscriber** (p. 1390) have a common partition name string.

Entity:

DDSPublisher (p. 1346), **DDSSubscriber** (p. 1390)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **YES** (p. 340)

6.91.2 Usage

This policy allows the introduction of a logical partition concept inside the 'physical' partition induced by a *domain*.

Usually DataReaders and DataWriters are matched only by their topic (so that data are only sent by DataWriters to DataReaders for the same topic). The Partition QoS policy allows you to add one or more strings, "partitions", to a Publisher and/or Subscriber. If partitions are added, then a DataWriter and DataReader for the same topic are only considered matched if their Publishers and Subscribers have partitions in common (intersecting partitions).

Since the set of partitions for a publisher or subscriber can be dynamically changed, the Partition QoS policy is useful to control which DataWriters can send data to which DataReaders and vice versa – even if all of the DataWriters and DataReaders are for the same topic. This facility is useful for creating temporary separation groups among entities that would otherwise be connected to and exchange data each other.

Failure to match partitions is not considered an incompatible QoS and does not trigger any listeners or conditions. A change in this policy *can* potentially modify the "match" of existing DataReader and DataWriter entities. It may establish new "matches" that did not exist before, or break existing matches.

Partition strings are usually directly matched via string comparisons. However, partition strings can also contain wildcard symbols so that partitions can be matched via pattern matching. As long as the partitions or wildcard patterns of a Publisher intersect with the partitions or wildcard patterns of a Subscriber, their DataWriters and DataReaders of the same topic are able to match; otherwise they are not.

These partition name patterns are regular expressions as defined by the POSIX fnmatch API (1003.2-1992 section B.6). Either **DDSPublisher** (p. 1346) or **DDSSubscriber** (p. 1390) may include regular expressions in partition names, but no two names that both contain wildcards will ever be considered to match. This means that although regular expressions may be used both at publisher as well as subscriber side, RTI Connext will not try to match two regular expressions (between publishers and subscribers).

Each publisher and subscriber must belong to at least one logical partition. A regular expression is not considered to be a logical partition. If a publisher or subscriber has not specify a logical partition, it is assumed to be in the default partition. The default partition is defined to be an empty string (""). Put another way:

- An empty sequence of strings in this QoS policy is considered equivalent to a sequence containing only a single string, the empty string.
- A string sequence that contains only regular expressions and no literal strings, it is treated as if it had an additional element, the empty string.

Partitions are different from creating **DDSEntity** (p. 1253) objects in different domains in several ways.

- First, entities belonging to different domains are completely isolated from each other; there is no traffic, meta-traffic or any other way for an application or RTI Connext itself to see entities in a domain it does not belong to.
- Second, a **DDSEntity** (p. 1253) can only belong to one domain whereas a **DDSEntity** (p. 1253) can be in multiple partitions.

- Finally, as far as RTI Connext is concerned, each unique data instance is identified by the tuple **(DomainID, DDSTopic (p. 1419), key)**. Therefore two **DDSEntity** (p. 1253) objects in different domains cannot refer to the same data instance. On the other hand, the same data instance can be made available (published) or requested (subscribed) on one or more partitions.

6.91.3 Member Data Documentation

6.91.3.1 struct DDS_StringSeq DDS_PartitionQosPolicy::name [read]

A list of partition names.

Several restrictions apply to the partition names in this sequence. A violation of one of the following rules will result in a **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) when setting a **DDSPublisher** (p. 1346)'s or **DDSSubscriber** (p. 1390)'s QoS.

- A partition name string cannot be NULL, nor can it contain the reserved comma character (',').

- The maximum number of partition name strings allowable in a **DDS_PartitionQosPolicy** (p. 820) is specified on a domain basis in **DDS_DomainParticipantResourceLimitsQosPolicy::max_partitions** (p. 607). The length of this sequence may not be greater than that value.

- The maximum cumulative length of all partition name strings in a **DDS_PartitionQosPolicy** (p. 820) is specified on a domain basis in **DDS_DomainParticipantResourceLimitsQosPolicy::max_partition_cumulative_characters** (p. 608).

The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

[**default**] Empty sequence (zero-length sequence). Since no logical partition is specified, RTI Connext will assume the entity to be in default partition (empty string partition "").

[**range**] List of partition name with above restrictions

6.92 DDS_PresentationQosPolicy Struct Reference

Specifies how the samples representing changes to data instances are presented to a subscribing application.

Public Attributes

- **DDS_PresentationQosPolicyAccessScopeKind access_scope**

  Determines the largest scope spanning the entities for which the order and coherency of changes can be preserved.

- **DDS_Boolean coherent_access**

  Specifies support for coherent *access. Controls whether coherent access is supported within the scope* `access_scope`.

- **DDS_Boolean ordered_access**

  Specifies support for ordered *access to the samples received at the subscription end. Controls whether ordered access is supported within the scope* `access_scope`.

6.92.1 Detailed Description

Specifies how the samples representing changes to data instances are presented to a subscribing application.

This QoS policy controls the extent to which changes to data instances can be made dependent on each other and also the kind of dependencies that can be propagated and maintained by RTI Connext. Specifically, this policy affects the application's ability to:

- specify and receive coherent changes to instances
- specify the relative order in which changes are presented

Entity:

DDSPublisher (p. 1346), **DDSSubscriber** (p. 1390)

Status:

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.92.2 Usage

A **DDSDataReader** (p. 1087) will usually receive data in the order that it was sent by a **DDSDataWriter** (p. 1113), and the data is presented to the **DDSDataReader** (p. 1087) as soon as the application receives the next expected value. However, sometimes, you may want a set of data for the same **DDSDataWriter** (p. 1113) to be presented to the **DDSDataReader** (p. 1087) only after *all* of the elements of the set have been received. Or you may want the data to be presented in a different order than that in which it was received. Specifically for keyed data, you may want the middleware to present the data in keyed – or *instance* – order, such that samples pertaining to the same instance are presented together.

The Presentation QoS policy allows you to specify different *scopes* of presentation: within a **DDSDataWriter** (p. 1113), across instances of a single **DDSDataWriter** (p. 1113), and even across multiple used by different writers of a publisher. It also controls whether or not a set of changes within the scope is delivered at the same time or can be delivered as soon as each element is received.

- `coherent_access` controls whether RTI Connext will preserve the groupings of changes made by a publishing application by means of the operations **DDSPublisher::begin_coherent_changes** (p. 1362) and **DDSPublisher::end_coherent_changes** (p. 1362).

- `ordered_access` controls whether RTI Connext will preserve the order of changes.

- `access_scope` controls the granularity of the other settings. See below:

If `coherent_access` is set, then the `access_scope` controls the maximum extent of coherent changes. The behavior is as follows:

- If `access_scope` is set to **DDS_INSTANCE_PRESENTATION_QOS** (p. 352) (the default), the use of **DDSPublisher::begin_coherent_changes** (p. 1362) and **DDSPublisher::end_coherent_changes** (p. 1362) has no effect on how the subscriber can access the data, because with the scope limited to each instance, changes to separate instances are considered independent and thus cannot be grouped into a coherent set.

- If `access_scope` is set to **DDS_TOPIC_PRESENTATION_QOS** (p. 352), then coherent changes (indicated by their enclosure within calls to **DDSPublisher::begin_coherent_changes** (p. 1362) and **DDSPublisher::end_coherent_changes** (p. 1362)) will be made available as such to each remote **DDSDataReader** (p. 1087) independently. That is, changes made to instances within each individual **DDSDataWriter** (p. 1113) will be available as coherent with respect to other changes to instances in that same **DDSDataWriter** (p. 1113), but will not be grouped with changes made to instances belonging to a different **DDSDataWriter** (p. 1113).

- If access_scope is set to **DDS_GROUP_PRESENTATION_QOS** (p. 352), then coherent changes made to instances through a **DDSDataWriter** (p. 1113) attached to a common **DDSPublisher** (p. 1346) are made available as a unit to remote subscribers. *(RTI does not currently support this access scope.)*

If `ordered_access` is set, then the `access_scope` controls the maximum extent for which order will be preserved by RTI Connext.

- If `access_scope` is set to **DDS_INSTANCE_PRESENTATION_QOS** (p. 352) (the lowest level), then changes to each instance are considered unordered relative to changes to any other instance. That means that changes (creations, deletions, modifications) made to two instances are not necessarily seen in the order they occur. This is the case even if it is the same application thread making the changes using the same **DDSDataWriter** (p. 1113).

- If `access_scope` is set to **DDS_TOPIC_PRESENTATION_QOS** (p. 352), changes (creations, deletions, modifications) made by a single **DDSDataWriter** (p. 1113) are made available to subscribers in the same order they occur. Changes made to instances though different **DDSDataWriter** (p. 1113) entities are not necessarily seen in the order they occur. This is the case, even if the changes are made by a single application thread using **DDSDataWriter** (p. 1113) objects attached to the same **DDSPublisher** (p. 1346).

- Finally, if access_scope is set to **DDS_GROUP_PRESENTATION_QOS** (p. 352), changes made to instances via **DDSDataWriter** (p. 1113) entities attached to the same **DDSPublisher** (p. 1346) object are made available to subscribers on the same order they occur.

Note that this QoS policy controls the scope at which related changes are made available to the subscriber. This means the subscriber **can** access the changes

in a coherent manner and in the proper order; however, it does not necessarily imply that the **DDSSubscriber** (p. 1390) **will** indeed access the changes in the correct order. For that to occur, the application at the subscriber end must use the proper logic in reading the **DDSDataReader** (p. 1087) objects.

For **DDS_GROUP_PRESENTATION_QOS** (p. 352) the subscribing application must use the APIs **DDSSubscriber::begin_access** (p. 1405), **DDSSubscriber::end_access** (p. 1406) and **DDSSubscriber::get_datareaders** (p. 1407) to access the changes in the proper order.

6.92.3 Compatibility

The value offered is considered compatible with the value requested if and only if the following conditions are met:

- the inequality *offered* `access_scope` >= *requested* `access_scope` evaluates to 'TRUE' or requested `access_scope` is **DDS_HIGHEST_OFFERED_PRESENTATION_QOS** (p. 352). For the purposes of this inequality, the values of `access_scope` are considered ordered such that **DDS_INSTANCE_PRESENTATION_QOS** (p. 352) < **DDS_TOPIC_PRESENTATION_QOS** (p. 352) < **DDS_GROUP_PRESENTATION_QOS** (p. 352).
- requested `coherent_access` is **DDS_BOOLEAN_FALSE** (p. 299), or else both offered and requested `coherent_access` are **DDS_BOOLEAN_TRUE** (p. 298).
- requested `ordered_access` is **DDS_BOOLEAN_FALSE** (p. 299), or else both offered and requested `ordered_access` are **DDS_BOOLEAN_TRUE** (p. 298).

6.92.4 Member Data Documentation

6.92.4.1 DDS_PresentationQosPolicyAccessScopeKind DDS_PresentationQosPolicy::access_scope

Determines the largest scope spanning the entities for which the order and coherency of changes can be preserved.

[default] DDS_INSTANCE_PRESENTATION_QOS (p. 352)

6.92.4.2 DDS_Boolean DDS_PresentationQosPolicy::coherent_access

Specifies support for *coherent* access. Controls whether coherent access is supported within the scope `access_scope`.

That is, the ability to group a set of changes as a unit on the publishing end such that they are received as a unit at the subscribing end.

Note: To use this feature, the DataWriter must be configured for RELIABLE communication (see **DDS_RELIABLE_RELIABILITY_QOS** (p. 363)).

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.92.4.3 DDS_Boolean DDS_PresentationQosPolicy::ordered_access

Specifies support for *ordered* access to the samples received at the subscription end. Controls whether ordered access is supported within the scope `access_scope`.

That is, the ability of the subscriber to see changes in the same order as they occurred on the publishing end.

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.93 DDS_ProductVersion_t Struct Reference

<<*eXtension*>> *(*p. *199)* Type used to represent the current version of RTI Connext.

Public Attributes

- **DDS_Char major**

  Major product version.

- **DDS_Char minor**

  Minor product version.

- **DDS_Char release**

  Release letter for product version.

- **DDS_Char revision**

  Revision number of product.

6.93.1 Detailed Description

<<*eXtension*>> *(*p. *199)* Type used to represent the current version of RTI Connext.

6.93.2 Member Data Documentation

6.93.2.1 DDS_Char DDS_ProductVersion_t::major

Major product version.

6.93.2.2 DDS_Char DDS_ProductVersion_t::minor

Minor product version.

6.93.2.3 DDS_Char DDS_ProductVersion_t::release

Release letter for product version.

6.93.2.4 DDS_Char DDS_ProductVersion_t::revision

Revision number of product.

6.94 DDS_ProfileQosPolicy Struct Reference

Configures the way that XML documents containing QoS profiles are loaded by RTI Connext.

Public Attributes

- struct **DDS_StringSeq string_profile**

  Sequence of strings containing a XML document to load.

- struct **DDS_StringSeq url_profile**

  *Sequence of **URL groups** (p. 153) containing a set of XML documents to load.*

- **DDS_Boolean ignore_user_profile**

  Ignores the file USER_QOS_PROFILES.xml in the current working directory.

- **DDS_Boolean ignore_environment_profile**

  *Ignores the value of the **NDDS_QOS_PROFILES environment variable** (p. 153).*

- **DDS_Boolean ignore_resource_profile**

  Ignores the file NDDS_QOS_PROFILES.xml under $NDDSHOME/resource/qos_profiles_4.4d/xml.

6.94.1 Detailed Description

Configures the way that XML documents containing QoS profiles are loaded by RTI Connext.

All QoS values for Entities can be configured in QoS profiles defined in XML documents. XML documents can be passed to RTI Connext in string form or, more likely, through files found on a file system.

There are also default locations where DomainParticipants will look for files to load QoS profiles. These include the current working directory from where an application is started, a file in the distribution directory for RTI Connext, and the locations specified by an environment variable. You may disable any or all of these default locations using the Profile QoS policy.

Entity:

DDSDomainParticipantFactory (p. 1216)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **Changeable** (p. 340)

6.94.2 Member Data Documentation

6.94.2.1 struct DDS_StringSeq DDS_ProfileQosPolicy::string_profile [read]

Sequence of strings containing a XML document to load.

The concatenation of the strings in this sequence must be a valid XML document according to the XML QoS profile schema.

[**default**] Empty sequence (zero-length).

6.94.2.2 struct DDS_StringSeq DDS_ProfileQosPolicy::url_profile [read]

Sequence of **URL groups** (p. 153) containing a set of XML documents to load.

Only one of the elements of each group will be loaded by RTI Connext, starting from the left.

[**default**] Empty sequence (zero-length).

6.94.2.3 DDS_Boolean DDS_ProfileQosPolicy::ignore_user_profile

Ignores the file USER_QOS_PROFILES.xml in the current working directory.

When this field is set to **DDS_BOOLEAN_TRUE** (p. 298), the QoS profiles contained in the file USER_QOS_PROFILES.xml in the current working directory will be ignored.

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.94.2.4 DDS_Boolean DDS_ProfileQosPolicy::ignore_environment_profile

Ignores the value of the **NDDS_QOS_PROFILES environment variable** (p. 153).

When this field is set to **DDS_BOOLEAN_TRUE** (p. 298), the value of the environment variable NDDS_QOS_PROFILES will be ignored.

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.94.2.5 DDS_Boolean DDS_ProfileQosPolicy::ignore_resource_profile

Ignores the file NDDS_QOS_PROFILES.xml under $NDDSHOME/resource/qos_profiles_4.4d/xml.

When this field is set to **DDS_BOOLEAN_TRUE** (p. 298), the QoS profiles contained in the file NDDS_QOS_PROFILES.xml under $NDDSHOME/resource/qos_profiles_5.0.0/xml will be ignored.

[**default**] **DDS_BOOLEAN_FALSE** (p. 299)

6.95 DDS_Property_t Struct Reference

Properties are name/value pairs objects.

Public Attributes

- char * **name**

  Property name.

- char * **value**

  Property value.

- **DDS_Boolean propagate**

  Indicates if the property must be propagated on discovery.

6.95.1 Detailed Description

Properties are name/value pairs objects.

6.95.2 Member Data Documentation

6.95.2.1 char* DDS_Property_t::name

Property name.

It must be a NULL-terminated string allocated with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459).

6.95.2.2 char* DDS_Property_t::value

Property value.

It must be a NULL-terminated string allocated with **DDS_String_alloc** (p. 458) or **DDS_String_dup** (p. 459).

6.95.2.3 DDS_Boolean DDS_Property_t::propagate

Indicates if the property must be propagated on discovery.

6.96 DDS_PropertyQosPolicy Struct Reference

Stores name/value(string) pairs that can be used to configure certain parameters of RTI Connext that are not exposed through formal QoS policies. Can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery.

Public Attributes

- struct **DDS_PropertySeq value**

  Sequence of properties.

6.96.1 Detailed Description

Stores name/value(string) pairs that can be used to configure certain parameters of RTI Connext that are not exposed through formal QoS policies. Can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery.

Entity:

DDSDomainParticipant (p. 1139) **DDSDataReader** (p. 1087) **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = N/A;
Changeable (p. 340) = **YES** (p. 340)

See also:

DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.96.2 Usage

The PROPERTY QoS policy can be used to associate a set of properties in the form of (name,value) pairs with a **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113), or **DDSDomainParticipant** (p. 1139). This is similar to the **DDS_UserDataQosPolicy** (p. 1048), except this policy uses (name, value) pairs, and you can select whether or not a particular pair should be propagated (included in the builtin topic).

This QoS policy may be used to configure:

- Durable Writer History, see **Configuring Durable Writer History** (p. 146)
- Durable Reader State, see **Configuring Durable Reader State** (p. 146)
- Builtin Transport Plugins, see **UDPv4 Transport Property Names in Property QoS Policy of Domain Participant** (p. 268), **UDPv6 Transport Property Names in Property QoS Policy of Domain Participant** (p. 278), and **Shared Memory Transport Property Names in Property QoS Policy of Domain Participant** (p. 261)
- Extension Transport Plugins, see **Loading Transport Plugins through Property QoS Policy of Domain Participant** (p. 131)
- **Clock Selection** (p. 29)

In addition, you may add your own name/value pairs to the Property QoS policy of an Entity. Via this QoS policy, you can direct RTI Connext to propagate these name/value pairs with the discovery information for the Entity. Applications that discover the Entity can then access the user-specific name/value pairs in the discovery information of the remote Entity. This allows you to add meta-information about an Entity for application-specific use, for example, authentication/authorization certificates (which can also be done using the **DDS_UserDataQosPolicy** (p. 1048) or **DDS_GroupDataQosPolicy** (p. 755)).

6.96.2.1 Reasons for Using the PropertyQosPolicy

- Supports dynamic loading of extension transports (such as RTI Secure WAN Transport)
- Supports multiple instances of the builtin transports
- Allows full pluggable transport configuration for non-C/C++ language bindings (Java, .NET, etc.)
- Avoids the process of creating entities disabled, changing their QoS settings, then enabling them
- Allows selection of clock

Some of the RTI Connext capabilities configurable via the Property QoS policy can also be configured in code via APIs. However, the Property QoS policy

allows you to configure those parameters via XML files. In addition, some of the configuration APIs will only work if the Entity was created in a disabled state and then enabled after the configuration change was applied. By configuring those parameters using the Property QoS policy during entity creation, you avoid the additional work of first creating a disabled entity and then enabling it afterwards.

There are helper functions to facilitate working with properties, see the DDS_PropertyQosPolicyHelper class on the **PROPERTY** (p. 436) page.

6.96.3 Member Data Documentation

6.96.3.1 struct DDS_PropertySeq DDS_PropertyQosPolicy::value [read]

Sequence of properties.

[**default**] An empty list.

6.97 DDS_PropertySeq Struct Reference

Declares IDL `sequence` < **DDS_Property_t** (p. 833) >.

6.97.1 Detailed Description

Declares IDL `sequence` < **DDS_Property_t** (p. 833) >.

See also:

DDS_Property_t (p. 833)

6.98 DDS_ProtocolVersion_t Struct Reference

<<*eXtension*>> *(p. 199)* Type used to represent the version of the RTPS protocol.

Public Attributes

- **DDS_Octet major**

  Major protocol version number.

- **DDS_Octet minor**

  Minor protocol version number.

6.98.1 Detailed Description

<<*eXtension*>> *(p. 199)* Type used to represent the version of the RTPS protocol.

6.98.2 Member Data Documentation

6.98.2.1 DDS_Octet DDS_ProtocolVersion_t::major

Major protocol version number.

6.98.2.2 DDS_Octet DDS_ProtocolVersion_t::minor

Minor protocol version number.

6.99 DDS_PublicationBuiltinTopicData Struct Reference

Entry created when a **DDSDataWriter** (p. 1113) is discovered in association with its Publisher.

Public Attributes

- **DDS_BuiltinTopicKey_t key**
  DCPS key to distinguish entries.
- **DDS_BuiltinTopicKey_t participant_key**
  DCPS key of the participant to which the DataWriter belongs.
- char * **topic_name**
  Name of the related ***DDSTopic*** *(p. 1419).*
- char * **type_name**
  Name of the type attached to the ***DDSTopic*** *(p. 1419).*
- struct **DDS_DurabilityQosPolicy durability**
  durability policy of the corresponding DataWriter
- struct **DDS_DurabilityServiceQosPolicy durability_service**
  durability_service policy of the corresponding DataWriter
- struct **DDS_DeadlineQosPolicy deadline**
  Policy of the corresponding DataWriter.
- struct **DDS_LatencyBudgetQosPolicy latency_budget**
  Policy of the corresponding DataWriter.
- struct **DDS_LivelinessQosPolicy liveliness**
  Policy of the corresponding DataWriter.
- struct **DDS_ReliabilityQosPolicy reliability**
  Policy of the corresponding DataWriter.
- struct **DDS_LifespanQosPolicy lifespan**
  Policy of the corresponding DataWriter.
- struct **DDS_UserDataQosPolicy user_data**

Policy of the corresponding DataWriter.

^ struct **DDS_OwnershipQosPolicy ownership**

Policy of the corresponding DataWriter.

^ struct **DDS_OwnershipStrengthQosPolicy ownership_strength**

Policy of the corresponding DataWriter.

^ struct **DDS_DestinationOrderQosPolicy destination_order**

Policy of the corresponding DataWriter.

^ struct **DDS_PresentationQosPolicy presentation**

Policy of the Publisher to which the DataWriter belongs.

^ struct **DDS_PartitionQosPolicy partition**

Policy of the Publisher to which the DataWriter belongs.

^ struct **DDS_TopicDataQosPolicy topic_data**

Policy of the related Topic.

^ struct **DDS_GroupDataQosPolicy group_data**

Policy of the Publisher to which the DataWriter belongs.

^ struct **DDS_TypeCode** * **type_code**

<<**eXtension**>> (p. 199) *Type code information of the corresponding Topic*

^ **DDS_BuiltinTopicKey_t publisher_key**

<<**eXtension**>> (p. 199) *DCPS key of the publisher to which the DataWriter belongs*

^ struct **DDS_PropertyQosPolicy property**

<<**eXtension**>> (p. 199) *Properties of the corresponding DataWriter.*

^ struct **DDS_LocatorSeq unicast_locators**

<<**eXtension**>> (p. 199) *Custom unicast locators that the endpoint can specify. The default locators will be used if this is not specified.*

^ struct **DDS_GUID_t virtual_guid**

<<**eXtension**>> (p. 199) *Virtual GUID associated to the DataWriter.*

^ **DDS_ProtocolVersion_t rtps_protocol_version**

<<**eXtension**>> (p. 199) *Version number of the RTPS wire protocol used.*

- struct **DDS_VendorId_t rtps_vendor_id**

  <<**eXtension**>> (p. 199) *ID of vendor implementing the RTPS wire protocol.*

- struct **DDS_ProductVersion_t product_version**

  <<**eXtension**>> (p. 199) *This is a vendor specific parameter. It gives the current version for rti-dds.*

- struct **DDS_LocatorFilterQosPolicy locator_filter**

  <<**eXtension**>> (p. 199) *Policy of the corresponding DataWriter*

- **DDS_Boolean disable_positive_acks**

  <<**eXtension**>> (p. 199) *This is a vendor specific parameter. Determines whether matching DataReaders send positive acknowledgements for reliability.*

- struct **DDS_EntityNameQosPolicy publication_name**

  <<**eXtension**>> (p. 199) *The publication name and role name.*

6.99.1 Detailed Description

Entry created when a **DDSDataWriter** (p. 1113) is discovered in association with its Publisher.

Data associated with the built-in topic **DDS_PUBLICATION_TOPIC_NAME** (p. 289). It contains QoS policies and additional information that apply to the remote **DDSDataWriter** (p. 1113) the related **DDSPublisher** (p. 1346).

See also:

DDS_PUBLICATION_TOPIC_NAME (p. 289)
DDSPublicationBuiltinTopicDataDataReader (p. 1344)

6.99.2 Member Data Documentation

6.99.2.1 DDS_BuiltinTopicKey_t DDS_PublicationBuiltinTopicData::key

DCPS key to distinguish entries.

6.99.2.2 DDS_BuiltinTopicKey_t DDS_PublicationBuiltinTopicData::participant_key

DCPS key of the participant to which the DataWriter belongs.

6.99.2.3 char* DDS_PublicationBuiltinTopicData::topic_name

Name of the related **DDSTopic** (p. 1419).

The length of this string is limited to 255 characters.

The memory for this field is managed as described in **Conventions** (p. 457).

See also:

Conventions (p. 457)

6.99.2.4 char* DDS_PublicationBuiltinTopicData::type_name

Name of the type attached to the **DDSTopic** (p. 1419).

The length of this string is limited to 255 characters.

The memory for this field is managed as described in **Conventions** (p. 457).

See also:

Conventions (p. 457)

6.99.2.5 struct DDS_DurabilityQosPolicy DDS_PublicationBuiltinTopicData::durability [read]

durability policy of the corresponding DataWriter

6.99.2.6 struct DDS_DurabilityServiceQosPolicy DDS_PublicationBuiltinTopicData::durability_service [read]

durability_service policy of the corresponding DataWriter

6.99.2.7 struct DDS_DeadlineQosPolicy DDS_-PublicationBuiltinTopicData::deadline [read]

Policy of the corresponding DataWriter.

6.99.2.8 struct DDS_LatencyBudgetQosPolicy DDS_PublicationBuiltinTopicData::latency_budget [read]

Policy of the corresponding DataWriter.

6.99.2.9 struct DDS_LivelinessQosPolicy DDS_-PublicationBuiltinTopicData::liveliness [read]

Policy of the corresponding DataWriter.

6.99.2.10 struct DDS_ReliabilityQosPolicy DDS_-PublicationBuiltinTopicData::reliability [read]

Policy of the corresponding DataWriter.

6.99.2.11 struct DDS_LifespanQosPolicy DDS_-PublicationBuiltinTopicData::lifespan [read]

Policy of the corresponding DataWriter.

6.99.2.12 struct DDS_UserDataQosPolicy DDS_-PublicationBuiltinTopicData::user_data [read]

Policy of the corresponding DataWriter.

6.99.2.13 struct DDS_OwnershipQosPolicy DDS_-PublicationBuiltinTopicData::ownership [read]

Policy of the corresponding DataWriter.

6.99.2.14 struct DDS_OwnershipStrengthQosPolicy DDS_PublicationBuiltinTopicData::ownership_strength [read]

Policy of the corresponding DataWriter.

6.99.2.15 struct DDS_DestinationOrderQosPolicy DDS_PublicationBuiltinTopicData::destination_order [read]

Policy of the corresponding DataWriter.

6.99.2.16 struct DDS_PresentationQosPolicy DDS_PublicationBuiltinTopicData::presentation [read]

Policy of the Publisher to which the DataWriter belongs.

6.99.2.17 struct DDS_PartitionQosPolicy DDS_PublicationBuiltinTopicData::partition [read]

Policy of the Publisher to which the DataWriter belongs.

6.99.2.18 struct DDS_TopicDataQosPolicy DDS_PublicationBuiltinTopicData::topic_data [read]

Policy of the related Topic.

6.99.2.19 struct DDS_GroupDataQosPolicy DDS_PublicationBuiltinTopicData::group_data [read]

Policy of the Publisher to which the DataWriter belongs.

6.99.2.20 struct DDS_TypeCode* DDS_PublicationBuiltinTopicData::type_code [read]

<<*eXtension*>> *(p. 199)* Type code information of the corresponding Topic

6.99.2.21 DDS_BuiltinTopicKey_t DDS_-PublicationBuiltinTopicData::publisher_key

<<*eXtension*>> *(*p. *199)* DCPS key of the publisher to which the DataWriter belongs

6.99.2.22 struct DDS_PropertyQosPolicy DDS_-PublicationBuiltinTopicData::property [read]

<<*eXtension*>> *(*p. *199)* Properties of the corresponding DataWriter.

6.99.2.23 struct DDS_LocatorSeq DDS_-PublicationBuiltinTopicData::unicast_locators [read]

<<*eXtension*>> *(*p. *199)* Custom unicast locators that the endpoint can specify. The default locators will be used if this is not specified.

6.99.2.24 struct DDS_GUID_t DDS_-PublicationBuiltinTopicData::virtual_guid [read]

<<*eXtension*>> *(*p. *199)* Virtual GUID associated to the DataWriter.

See also:

DDS_GUID_t (p. 757)

6.99.2.25 DDS_ProtocolVersion_t DDS_-PublicationBuiltinTopicData::rtps_protocol_version

<<*eXtension*>> *(*p. *199)* Version number of the RTPS wire protocol used.

6.99.2.26 struct DDS_VendorId_t DDS_-PublicationBuiltinTopicData::rtps_vendor_id [read]

<<*eXtension*>> *(*p. *199)* ID of vendor implementing the RTPS wire protocol.

6.99.2.27 struct DDS_ProductVersion_t DDS_PublicationBuiltinTopicData::product_version [read]

<<*eXtension*>> *(*p. *199)* This is a vendor specific parameter. It gives the current version for rti-dds.

6.99.2.28 struct DDS_LocatorFilterQosPolicy DDS_PublicationBuiltinTopicData::locator_filter [read]

<<*eXtension*>> *(*p. *199)* Policy of the corresponding DataWriter

Related to **DDS_MultiChannelQosPolicy** (p. 796).

6.99.2.29 DDS_Boolean DDS_PublicationBuiltinTopicData::disable_positive_acks

<<*eXtension*>> *(*p. *199)* This is a vendor specific parameter. Determines whether matching DataReaders send positive acknowledgements for reliability.

6.99.2.30 struct DDS_EntityNameQosPolicy DDS_PublicationBuiltinTopicData::publication_name [read]

<<*eXtension*>> *(*p. *199)* The publication name and role name.

This member contains the name and the role name of the discovered publication.

6.100 DDS_PublicationBuiltinTopicDataSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_PublicationBuiltinTopicData** (p. 839) > .

6.100.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_PublicationBuiltinTopicData** (p. 839) > .

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_PublicationBuiltinTopicData (p. 839)

6.101 DDS_PublicationMatchedStatus Struct Reference

DDS_PUBLICATION_MATCHED_STATUS (p. 325)

Public Attributes

- **DDS_Long total_count**

  *The total cumulative number of times the concerned **DDSDataWriter** (p. 1113) discovered a "match" with a **DDSDataReader** (p. 1087).*

- **DDS_Long total_count_change**

  The incremental changes in total_count since the last time the listener was called or the status was read.

- **DDS_Long current_count**

  *The current number of readers with which the **DDSDataWriter** (p. 1113) is matched.*

- **DDS_Long current_count_peak**

  <<**eXtension**>> (p. 199) *The highest value that current_count has reached until now.*

- **DDS_Long current_count_change**

  The change in current_count since the last time the listener was called or the status was read.

- **DDS_InstanceHandle_t last_subscription_handle**

  *A handle to the last **DDSDataReader** (p. 1087) that caused the the **DDSDataWriter** (p. 1113)'s status to change.*

6.101.1 Detailed Description

DDS_PUBLICATION_MATCHED_STATUS (p. 325)

A "match" happens when the **DDSDataWriter** (p. 1113) finds a **DDSDataReader** (p. 1087) for the same **DDSTopic** (p. 1419) and common partition with a requested QoS that is compatible with that offered by the **DDSDataWriter** (p. 1113).

This status is also changed (and the listener, if any, called) when a match is ended. A local **DDSDataWriter** (p. 1113) will become "unmatched" from

a remote **DDSDataReader** (p. 1087) when that **DDSDataReader** (p. 1087) goes away for any reason.

6.101.2 Member Data Documentation

6.101.2.1 DDS_Long DDS_PublicationMatchedStatus::total_count

The total cumulative number of times the concerned **DDSDataWriter** (p. 1113) discovered a "match" with a **DDSDataReader** (p. 1087).

This number increases whenever a new match is discovered. It does not change when an existing match goes away.

6.101.2.2 DDS_Long DDS_PublicationMatchedStatus::total_count_change

The incremental changes in total_count since the last time the listener was called or the status was read.

6.101.2.3 DDS_Long DDS_PublicationMatchedStatus::current_count

The current number of readers with which the **DDSDataWriter** (p. 1113) is matched.

This number increases when a new match is discovered and decreases when an existing match goes away.

6.101.2.4 DDS_Long DDS_PublicationMatchedStatus::current_count_peak

*<<**eXtension**>>* *(p. 199)* The highest value that current_count has reached until now.

6.101.2.5 DDS_Long DDS_PublicationMatchedStatus::current_count_change

The change in current_count since the last time the listener was called or the status was read.

6.101.2.6 DDS_InstanceHandle_t DDS_PublicationMatchedStatus::last_subscription_handle

A handle to the last **DDSDataReader** (p. 1087) that caused the the **DDSDataWriter** (p. 1113)'s status to change.

6.102 DDS_PublisherQos Struct Reference

QoS policies supported by a **DDSPublisher** (p. 1346) entity.

Public Attributes

- struct **DDS_PresentationQosPolicy presentation**

  Presentation policy, ***PRESENTATION*** *(p. 351).*

- struct **DDS_PartitionQosPolicy partition**

  Partition policy, ***PARTITION*** *(p. 361).*

- struct **DDS_GroupDataQosPolicy group_data**

  Group data policy, ***GROUP_DATA*** *(p. 347).*

- struct **DDS_EntityFactoryQosPolicy entity_factory**

  Entity factory policy, ***ENTITY_FACTORY*** *(p. 377).*

- struct **DDS_AsynchronousPublisherQosPolicy asynchronous_publisher**

  <<**eXtension**>> (p. 199) *Asynchronous publishing settings for the* ***DDSPublisher*** *(p. 1346) and all entities that are created by it.*

- struct **DDS_ExclusiveAreaQosPolicy exclusive_area**

  <<**eXtension**>> (p. 199) *Exclusive area for the* ***DDSPublisher*** *(p. 1346) and all entities that are created by it.*

6.102.1 Detailed Description

QoS policies supported by a **DDSPublisher** (p. 1346) entity.

You must set certain members in a consistent manner:

length of DDS_PublisherQos::group_data.value <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .publisher_group_data_max_length

length of DDS_PublisherQos::partition.name <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .max_partitions

combined number of characters (including terminating 0) in DDS_PublisherQos::partition.name <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .max_partition_cumulative_characters

If any of the above are not true, **DDSPublisher::set_qos** (p. 1366) and **DDSPublisher::set_qos_with_profile** (p. 1366) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) and **DDSDomainParticipant::create_publisher** (p. 1169) will return NULL.

6.102.2 Member Data Documentation

6.102.2.1 struct DDS_PresentationQosPolicy DDS_PublisherQos::presentation [read]

Presentation policy, **PRESENTATION** (p. 351).

6.102.2.2 struct DDS_PartitionQosPolicy DDS_PublisherQos::partition [read]

Partition policy, **PARTITION** (p. 361).

6.102.2.3 struct DDS_GroupDataQosPolicy DDS_PublisherQos::group_data [read]

Group data policy, **GROUP_DATA** (p. 347).

6.102.2.4 struct DDS_EntityFactoryQosPolicy DDS_PublisherQos::entity_factory [read]

Entity factory policy, **ENTITY_FACTORY** (p. 377).

6.102.2.5 struct DDS_AsynchronousPublisherQosPolicy DDS_PublisherQos::asynchronous_publisher [read]

<<***eXtension***>> *(p. 199)* Asynchronous publishing settings for the **DDSPublisher** (p. 1346) and all entities that are created by it.

6.102.2.6 struct DDS_ExclusiveAreaQosPolicy DDS_PublisherQos::exclusive_area [read]

<<***eXtension***>> *(p. 199)* Exclusive area for the **DDSPublisher** (p. 1346) and all entities that are created by it.

6.103 DDS_PublishModeQosPolicy Struct Reference

Specifies how RTI Connext sends application data on the network. This QoS policy can be used to tell RTI Connext to use its *own* thread to send data, instead of the user thread.

Public Attributes

- **DDS_PublishModeQosPolicyKind kind**

  Publishing mode.

- char * **flow_controller_name**

  Name of the associated flow controller.

- **DDS_Long priority**

  Publication priority.

6.103.1 Detailed Description

Specifies how RTI Connext sends application data on the network. This QoS policy can be used to tell RTI Connext to use its *own* thread to send data, instead of the user thread.

The publishing mode of a **DDSDataWriter** (p. 1113) determines whether data is written synchronously in the context of the user thread when calling **FooDataWriter::write** (p. 1484) or asynchronously in the context of a separate thread internal to the middleware.

Each **DDSPublisher** (p. 1346) spawns a single asynchronous publishing thread (**DDS_AsynchronousPublisherQosPolicy::thread** (p. 468)) to serve all its asynchronous **DDSDataWriter** (p. 1113) instances.

See also:

DDS_AsynchronousPublisherQosPolicy (p. 466)
DDS_HistoryQosPolicy (p. 758)
DDSFlowController (p. 1259)

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.103.2 Usage

The fastest way for RTI Connext to send data is for the user thread to execute the middleware code that actually sends the data itself. However, there are times when user applications may need or want an internal middleware thread to send the data instead. For instance, to send large data reliably, you must use an asynchronous thread.

When data is written asynchronously, a **DDSFlowController** (p. 1259), identified by `flow_controller_name`, can be used to shape the network traffic. Shaping a data flow usually means limiting the maximum data rates at which the middleware will send data for a **DDSDataWriter** (p. 1113). The flow controller will buffer any excess data and only send it when the send rate drops below the maximum rate. The flow controller's properties determine when the asynchronous publishing thread is allowed to send data and how much.

Asynchronous publishing may increase latency, but offers the following advantages:

- The **FooDataWriter::write** (p. 1484) call does not make any network calls and is therefore faster and more deterministic. This becomes important when the user thread is executing time-critical code.

- When data is written in bursts or when sending large data types as multiple fragments, a flow controller can throttle the send rate of the asynchronous publishing thread to avoid flooding the network.

- Asynchronously written samples for the same destination will be coalesced into a single network packet which reduces bandwidth consumption.

The maximum number of samples that will be coalesced depends on **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525) (each sample requires at least 2-4 gather-send buffers). Performance can be improved by increasing **NDDS_Transport_Property_t::gather_send_buffer_count_max** (p. 1525). Note that the maximum value is operating system dependent.

The middleware must queue samples until they can be sent by the asynchronous publishing thread (as determined by the corresponding **DDSFlowController** (p. 1259)). The number of samples that will be queued is determined by the **DDS_HistoryQosPolicy** (p. 758). When using **DDS_KEEP_LAST_HISTORY_QOS** (p. 368), only the most recent **DDS_HistoryQosPolicy::depth** (p. 760) samples are kept in the queue. Once unsent samples are removed from the queue, they are no longer available to the

asynchronous publishing thread and will therefore never be sent.

6.103.3 Member Data Documentation

6.103.3.1 DDS_PublishModeQosPolicyKind DDS_PublishModeQosPolicy::kind

Publishing mode.

[default] **DDS_SYNCHRONOUS_PUBLISH_MODE_QOS** (p. 421)

6.103.3.2 char* DDS_PublishModeQosPolicy::flow_controller_name

Name of the associated flow controller.

NULL value or zero-length string refers to **DDS_DEFAULT_FLOW_CONTROLLER_NAME** (p. 91).

Unless `flow_controller_name` points to a built-in flow controller, finalizing the **DDS_DataWriterQos** (p. 553) will also free the string pointed to by `flow_controller_name`. Therefore, use **DDS_String_dup** (p. 459) before passing the string to `flow_controller_name`, or reset `flow_controller_name` to NULL before finalizing the QoS.

Please refer to **Conventions** (p. 457) for more information.

See also:

DDSDomainParticipant::create_flowcontroller (p. 1184)
DDS_DEFAULT_FLOW_CONTROLLER_NAME (p. 91)
DDS_FIXED_RATE_FLOW_CONTROLLER_NAME (p. 92)
DDS_ON_DEMAND_FLOW_CONTROLLER_NAME (p. 93)

[default] **DDS_DEFAULT_FLOW_CONTROLLER_NAME** (p. 91)

6.103.3.3 DDS_Long DDS_PublishModeQosPolicy::priority

Publication priority.

A positive integer value designating the relative priority of the **DDSDataWriter** (p. 1113), used to determine the transmission order of pending writes.

Use of publication priorities requires the asynchronous publisher (**DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422)) with **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) set to **DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90).

Larger numbers have higher priority.

For multi-channel DataWriters, if the publication priority of any channel is set to any value other than **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), then the channel's priority will take precedence over that of the DataWriter.

For multi-channel DataWriters, if the publication priority of any channel is **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), then the channel will inherit the publication priority of the DataWriter.

If the publicaton priority of the DataWriter, and of any channel of a multi-channel DataWriter, are **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), then the priority of the DataWriter or DataWriter channel will be assigned the lowest priority value.

If the publication priority of the DataWriter is **DDS_PUBLICATION_PRIORITY_AUTOMATIC** (p. 421), then the DataWriter will be assigned the priority of the largest publication priority of all samples in the DataWriter.

The publication priority of each sample can be set in the **DDS_WriteParams_t** (p. 1067) of the **FooDataWriter::write_w_params** (p. 1487) function.

For dispose and unregister samples, use the **DDS_WriteParams_t** (p. 1067) of **FooDataWriter::dispose_w_params** (p. 1491) and **FooDataWriter::unregister_instance_w_params** (p. 1483).

[default] DDS_PUBLICATION_PRIORITY_UNDEFINED (p. 421)

6.104 DDS_QosPolicyCount Struct Reference

Type to hold a counter for a **DDS_QosPolicyId_t** (p. 341).

Public Attributes

- **DDS_QosPolicyId_t policy_id**
  The `QosPolicy` *ID.*
- **DDS_Long count**
  a counter

6.104.1 Detailed Description

Type to hold a counter for a **DDS_QosPolicyId_t** (p. 341).

6.104.2 Member Data Documentation

6.104.2.1 DDS_QosPolicyId_t DDS_QosPolicyCount::policy_id

The `QosPolicy` ID.

6.104.2.2 DDS_Long DDS_QosPolicyCount::count

a counter

6.105 DDS_QosPolicyCountSeq Struct Reference

Declares IDL `sequence` < **DDS_QosPolicyCount** (p. 857) >.

6.105.1 Detailed Description

Declares IDL `sequence` < **DDS_QosPolicyCount** (p. 857) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_QosPolicyCount (p. 857)

6.106 DDS_ReaderDataLifecycleQosPolicy Struct Reference

Controls how a DataReader manages the lifecycle of the data that it has received.

Public Attributes

- struct **DDS_Duration_t autopurge_nowriter_samples_delay**

  *Maximum duration for which the **DDSDataReader** (p. 1087) will maintain information regarding an instance once its* `instance_state` *becomes **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117).*

- struct **DDS_Duration_t autopurge_disposed_samples_delay**

  *Maximum duration for which the **DDSDataReader** (p. 1087) will maintain samples for an instance once its* `instance_state` *becomes **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117).*

6.106.1 Detailed Description

Controls how a DataReader manages the lifecycle of the data that it has received.

When a DataReader receives data, it is stored in a receive queue for the DataReader. The user application may either take the data from the queue or leave it there.

This QoS policy controls whether or not RTI Connext will automatically remove data from the receive queue (so that user applications cannot access it afterwards) when it detects that there are no more DataWriters alive for that data. It specifies how long a **DDSDataReader** (p. 1087) must retain information regarding instances that have the instance_state **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117).

Note: This policy is not concerned with keeping reliable reader state or discovery information.

The **DDSDataReader** (p. 1087) internally maintains the samples that have not been "taken" by the application, subject to the constraints imposed by other QoS policies such as **DDS_HistoryQosPolicy** (p. 758) and **DDS_ResourceLimitsQosPolicy** (p. 879).

The **DDSDataReader** (p. 1087) also maintains information regarding the identity, `view_state` and `instance_state` of data instances even after all samples have been taken. This is needed to properly compute the states when future samples arrive.

Under normal circumstances the **DDSDataReader** (p. 1087) can only reclaim all resources for instances for which there are no writers and for which all samples have been 'taken'. The last sample the **DDSDataReader** (p. 1087) will have taken for that instance will have an `instance_state` of either **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117) or **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) depending on whether or not the last writer that had ownership of the instance disposed it.

In the absence of **READER_DATA_LIFECYCLE** (p. 376), this behavior could cause problems if the application forgets to take those samples. "Untaken" samples will prevent the **DDSDataReader** (p. 1087) from reclaiming the resources and they would remain in the **DDSDataReader** (p. 1087) indefinitely.

For keyed Topics, the consideration of removing data samples from the receive queue is done on a per instance (key) basis. Thus when RTI Connext detects that there are no longer DataWriters alive for a certain key value of a Topic (an instance of the Topic), it can be configured to remove all data samples for that instance (key).

Entity:

DDSDataReader (p. 1087)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **YES** (p. 340)

6.106.2 Member Data Documentation

6.106.2.1 struct DDS_Duration_t DDS_ReaderDataLifecycleQosPolicy::autopurge_nowriter_samples_delay [read]

Maximum duration for which the **DDSDataReader** (p. 1087) will maintain information regarding an instance once its `instance_state` becomes **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117).

After this time elapses, the **DDSDataReader** (p. 1087) will purge all internal information regarding the instance, any "untaken" samples will also be lost.

[default] **DDS_DURATION_INFINITE** (p. 305)

[range] [1 nanosec, 1 year] or **DDS_DURATION_INFINITE** (p. 305)

6.106.2.2 struct DDS_Duration_t DDS_ReaderDataLifecycleQosPolicy::autopurge_disposed_samples_delay [read]

Maximum duration for which the **DDSDataReader** (p. 1087) will maintain samples for an instance once its `instance_state` becomes **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117).

After this time elapses, the **DDSDataReader** (p. 1087) will purge all samples for the instance.

[default] **DDS_DURATION_INFINITE** (p. 305)

[range] [1 nanosec, 1 year] or **DDS_DURATION_INFINITE** (p. 305)

6.107 DDS_ReceiverPoolQosPolicy Struct Reference

Configures threads used by RTI Connext to receive and process data from transports (for example, UDP sockets).

Public Attributes

- struct **DDS_ThreadSettings_t thread**

  Receiver pool thread(s).

- **DDS_Long buffer_size**

  The receive buffer size.

- **DDS_Long buffer_alignment**

  The receive buffer alignment.

6.107.1 Detailed Description

Configures threads used by RTI Connext to receive and process data from transports (for example, UDP sockets).

This QoS policy is an extension to the DDS standard.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

See also:

Controlling CPU Core Affinity for RTI Threads (p. 330)

6.107.2 Usage

This QoS policy sets the thread properties such as priority level and stack size for the threads used by the middleware to receive and process data from transports.

RTI uses a separate receive thread per port per transport plug-in. To force RTI Connext to use a separate thread to process the data for a **DDSDataReader** (p. 1087), set a unique port for the **DDS_TransportUnicastQosPolicy** (p. 987) or **DDS_TransportMulticastQosPolicy** (p. 978) for the **DDSDataReader** (p. 1087).

This QoS policy also sets the size of the buffer used to store packets received from a transport. This buffer size will limit the largest single packet of data that a **DDSDomainParticipant** (p. 1139) will accept from a transport. Users will often set this size to the largest packet that any of the transports used by their application will deliver. For many applications, the value 65,536 (64 K) is a good choice; this value is the largest packet that can be sent/received via UDP.

6.107.3 Member Data Documentation

6.107.3.1 struct DDS_ThreadSettings_t DDS_ReceiverPoolQosPolicy::thread [read]

Receiver pool thread(s).

There is at least one receive thread, possibly more.

[**default**] priority above normal.

The actual value depends on your architecture:

For Windows: 2

For Solaris: OS default priority

For Linux: OS default priority

For LynxOS: 29

For INTEGRITY: 100

For VxWorks: 71

For all others: OS default priority.

[**default**] The actual value depends on your architecture:

For Windows: OS default stack size

For Solaris: OS default stack size

For Linux: OS default stack size

For LynxOS: 4∗16∗1024

For INTEGRITY: 4∗20∗1024

For VxWorks: 4∗16∗1024

For all others: OS default stack size.

[default] mask **DDS_THREAD_SETTINGS_FLOATING_POINT** (p. 329) | **DDS_THREAD_SETTINGS_STDIO** (p. 329)

6.107.3.2 DDS_Long DDS_ReceiverPoolQosPolicy::buffer_size

The receive buffer size.

The receive buffer is used by the receive thread to store the raw data that arrives over the transport.

In many applications, users will change the configuration of the built-in transport **NDDS_Transport_Property_t::message_size_max** (p. 1525) to increase the size of the largest data packet that can be sent or received through the transport. Typically, users will change the UDPv4 transport plugin's **NDDS_Transport_Property_t::message_size_max** (p. 1525) to 65536 (64 K), which is the largest packet that can be sent/received via UDP.

Ihe ReceiverPoolQosPolicy's `buffer_size` should be set to be the same value as the maximum **NDDS_Transport_Property_t::message_size_max** (p. 1525) across *all* of the transports being used.

If you are using the default configuration of the built-in transports, you should not need to change this buffer size.

In addition, if your application *only* uses transports that support zero-copy, then you do not need to modify the value of buffer_size, even if the **NDDS_Transport_Property_t::message_size_max** (p. 1525) of the transport is changed. Transports that support zero-copy do not copy their data into the buffer provided by the receive thread. Instead, they provide the receive thread data in a buffer allocated by the transport itself. The only built-in transport that supports zero-copy is the UDPv4 transport on VxWorks platforms.

[default] 9216

[range] [1, 1 GB]

6.107.3.3 DDS_Long DDS_ReceiverPoolQosPolicy::buffer_alignment

The receive buffer alignment.

Most users will not need to change this alignment.

[default] 16

[range] [1,1024] Value must be a power of 2.

6.108 DDS_ReliabilityQosPolicy Struct Reference

Indicates the level of reliability offered/requested by RTI Connext.

Public Attributes

- **DDS_ReliabilityQosPolicyKind kind**
  Kind of reliability.
- struct **DDS_Duration_t max_blocking_time**
  The maximum time a writer may block on a write() call.
- **DDS_ReliabilityQosPolicyAcknowledgmentModeKind acknowledgment_kind**
  Kind of reliable acknowledgment.

6.108.1 Detailed Description

Indicates the level of reliability offered/requested by RTI Connext.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323)

Properties:

RxO (p. 340) = YES
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.108.2 Usage

This policy indicates the level of reliability requested by a **DDSDataReader** (p. 1087) or offered by a **DDSDataWriter** (p. 1113).

The reliability of a connection between a DataWriter and DataReader is entirely user configurable. It can be done on a per DataWriter/DataReader connection.

A connection may be configured to be "best effort" which means that RTI Connext will not use any resources to monitor or guarantee that the data sent by a DataWriter is received by a DataReader.

For some use cases, such as the periodic update of sensor values to a GUI displaying the value to a person, **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363) delivery is often good enough. It is certainly the fastest, most efficient, and least resource-intensive (CPU and network bandwidth) method of getting the newest/latest value for a topic from DataWriters to DataReaders. But there is no guarantee that the data sent will be received. It may be lost due to a variety of factors, including data loss by the physical transport such as wireless RF or even Ethernet.

However, there are data streams (topics) in which you want an absolute guarantee that all data sent by a DataWriter is received reliably by DataReaders. This means that RTI Connext must check whether or not data was received, and repair any data that was lost by resending a copy of the data as many times as it takes for the DataReader to receive the data. RTI Connext uses a reliability protocol configured and tuned by these QoS policies: **DDS_HistoryQosPolicy** (p. 758), **DDS_DataWriterProtocolQosPolicy** (p. 535), **DDS_DataReaderProtocolQosPolicy** (p. 501), and **DDS_ResourceLimitsQosPolicy** (p. 879).

The Reliability QoS policy is simply a switch to turn on the reliability protocol for a DataWriter/DataReader connection. The level of reliability provided by RTI Connext is determined by the configuration of the aforementioned QoS policies.

You can configure RTI Connext to deliver *all* data in the order they were sent (also known as absolute or strict reliability). Or, as a tradeoff for less memory, CPU, and network usage, you can choose a reduced level of reliability where only the last N values are guaranteed to be delivered reliably to DataReaders (where N is user-configurable). In the reduced level of reliability, there are no guarantees that the data sent before the last N are received. Only the last N data packets are monitored and repaired if necessary.

These levels are ordered, **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363) < **DDS_RELIABLE_RELIABILITY_QOS** (p. 363). A **DDSDataWriter** (p. 1113) offering one level is implicitly offering all levels below.

Note: To send *large* data reliably, you will also need to set **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422). *Large* in this context means that the data cannot be sent as a single packet by the transport (for example, data larger than 63K when using UDP/IP).

The setting of this policy has a dependency on the setting of the **RESOURCE_LIMITS** (p. 371) policy. In case the reliability kind is set to **DDS_RELIABLE_RELIABILITY_QOS** (p. 363) the write operation on the **DDSDataWriter** (p. 1113) may block if the modification would cause data

to be lost or else cause one of the limits in specified in the **RESOURCE_LIMITS** (p. 371) to be exceeded. Under these circumstances, the **RELIABILITY** (p. 362) `max_blocking_time` configures the maximum duration the write operation may block.

If the **DDS_ReliabilityQosPolicy::kind** (p. 868) is set to **DDS_RELIABLE_RELIABILITY_QOS** (p. 363), data samples originating from a single **DDSDataWriter** (p. 1113) cannot be made available to the **DDSDataReader** (p. 1087) if there are previous data samples that have not been received yet due to a communication error. In other words, RTI Connext will repair the error and resend data samples as needed in order to reconstruct a correct snapshot of the **DDSDataWriter** (p. 1113) history before it is accessible by the **DDSDataReader** (p. 1087).

If the **DDS_ReliabilityQosPolicy::kind** (p. 868) is set to **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363), the service will not re-transmit missing data samples. However, for data samples originating from any one DataWriter the service will ensure they are stored in the **DDSDataReader** (p. 1087) history in the same order they originated in the **DDSDataWriter** (p. 1113). In other words, the **DDSDataReader** (p. 1087) may miss some data samples, but it will never see the value of a data object change from a newer value to an older value.

See also:

DDS_HistoryQosPolicy (p. 758)
DDS_ResourceLimitsQosPolicy (p. 879)

6.108.3 Compatibility

The value offered is considered compatible with the value requested if and only if:

- the inequality *offered* `kind` >= *requested* `kind` evaluates to 'TRUE'. For the purposes of this inequality, the values of **DDS_ReliabilityQosPolicy::kind** (p. 868) are considered ordered such that **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363) < **DDS_RELIABLE_RELIABILITY_QOS** (p. 363).

- The offered acknowledgment_kind = **DDS_PROTOCOL_ACKNOWLEDGMENT_MODE** (p. 363) and requested acknowledgment_kind = **DDS_PROTOCOL_ACKNOWLEDGMENT_MODE** (p. 363) OR offered acknowledgment_kind != **DDS_PROTOCOL_ACKNOWLEDGMENT_MODE** (p. 363).

6.108.4 Member Data Documentation

6.108.4.1 DDS_ReliabilityQosPolicyKind DDS_ReliabilityQosPolicy::kind

Kind of reliability.

[default] **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363) for **DDSDataReader** (p. 1087) and **DDSTopic** (p. 1419), **DDS_RELIABLE_RELIABILITY_QOS** (p. 363) for **DDSDataWriter** (p. 1113)

6.108.4.2 struct DDS_Duration_t DDS_ReliabilityQosPolicy::max_blocking_time [read]

The maximum time a writer may block on a write() call.

This setting applies only to the case where **DDS_ReliabilityQosPolicy::kind** (p. 868) = **DDS_RELIABLE_RELIABILITY_QOS** (p. 363). **FooDataWriter::write** (p. 1484) is allowed to block if the **DDSDataWriter** (p. 1113) does not have space to store the value written. Only applies to **DDSDataWriter** (p. 1113).

[default] 100 milliseconds

[range] [0,1 year] or **DDS_DURATION_INFINITE** (p. 305)

See also:

DDS_ResourceLimitsQosPolicy (p. 879)

6.108.4.3 DDS_ReliabilityQosPolicyAcknowledgmentModeKind DDS_ReliabilityQosPolicy::acknowledgment_kind

Kind of reliable acknowledgment.

This setting applies only to the case where **DDS_ReliabilityQosPolicy::kind** (p. 868) = **DDS_RELIABLE_RELIABILITY_QOS** (p. 363).

Sets the kind acknowledgments supported by a **DDSDataWriter** (p. 1113) and sent by **DDSDataReader** (p. 1087).

[default] **DDS_PROTOCOL_ACKNOWLEDGMENT_MODE** (p. 363)

6.109 DDS_ReliableReaderActivityChangedStatus Struct Reference

<<*eXtension*>> *(p. 199)* Describes the activity (i.e. are acknowledgements forthcoming) of reliable readers matched to a reliable writer.

Public Attributes

- **DDS_Long active_count**

  The current number of reliable readers currently matched with this reliable writer.

- **DDS_Long inactive_count**

  The number of reliable readers that have been dropped by this reliable writer because they failed to send acknowledgements in a timely fashion.

- **DDS_Long active_count_change**

  The most recent change in the number of active remote reliable readers.

- **DDS_Long inactive_count_change**

  The most recent change in the number of inactive remote reliable readers.

- **DDS_InstanceHandle_t last_instance_handle**

  The instance handle of the last reliable remote reader to be determined inactive.

6.109.1 Detailed Description

<<*eXtension*>> *(p. 199)* Describes the activity (i.e. are acknowledgements forthcoming) of reliable readers matched to a reliable writer.

Entity:

DDSDataWriter (p. 1113)

Listener:

DDSDataWriterListener (p. 1133)

This status is the reciprocal status to the **DDS_LivelinessChangedStatus** (p. 775) on the reader. It is different than the **DDS_LivelinessLostStatus**

(p. 777) on the writer in that the latter informs the writer about its own liveliness; this status informs the writer about the "liveliness" (activity) of its matched readers.

All counts in this status will remain at zero for best effort writers.

6.109.2 Member Data Documentation

6.109.2.1 DDS_Long DDS_-ReliableReaderActivityChangedStatus::active_count

The current number of reliable readers currently matched with this reliable writer.

6.109.2.2 DDS_Long DDS_-ReliableReaderActivityChangedStatus::inactive_count

The number of reliable readers that have been dropped by this reliable writer because they failed to send acknowledgements in a timely fashion.

A reader is considered to be inactive after is has been sent heartbeats **DDS_-RtpsReliableWriterProtocol_t::max_heartbeat_retries** (p. 895) times, each heartbeat having been separated from the previous by the current heartbeat period.

6.109.2.3 DDS_Long DDS_-ReliableReaderActivityChangedStatus::active_count_-change

The most recent change in the number of active remote reliable readers.

6.109.2.4 DDS_Long DDS_-ReliableReaderActivityChangedStatus::inactive_count_-change

The most recent change in the number of inactive remote reliable readers.

6.109.2.5 DDS_InstanceHandle_t DDS_-ReliableReaderActivityChangedStatus::last_instance_-handle

The instance handle of the last reliable remote reader to be determined inactive.

6.110 DDS_ReliableWriterCacheChangedStatus Struct Reference

<<*eXtension*>> *(p. 199)* A summary of the state of a data writer's cache of unacknowledged samples written.

Public Attributes

- struct **DDS_ReliableWriterCacheEventCount empty_reliable_writer_cache**

  The number of times the reliable writer's cache of unacknowledged samples has become empty.

- struct **DDS_ReliableWriterCacheEventCount full_reliable_writer_cache**

  The number of times the reliable writer's cache, or send window, of unacknowledged samples has become full.

- struct **DDS_ReliableWriterCacheEventCount low_watermark_reliable_writer_cache**

  The number of times the reliable writer's cache of unacknowledged samples has fallen to the low watermark.

- struct **DDS_ReliableWriterCacheEventCount high_watermark_reliable_writer_cache**

  The number of times the reliable writer's cache of unacknowledged samples has risen to the high watermark.

- **DDS_Long unacknowledged_sample_count**

  The current number of unacknowledged samples in the writer's cache.

- **DDS_Long unacknowledged_sample_count_peak**

  The highest value that unacknowledged_sample_count has reached until now.

6.110.1 Detailed Description

<<*eXtension*>> *(p. 199)* A summary of the state of a data writer's cache of unacknowledged samples written.

Entity:

DDSDataWriter (p. 1113)

Listener:

DDSDataWriterListener (p. 1133)

A written sample is unacknowledged (and therefore accounted for in this status) if the writer is reliable and one or more readers matched with the writer has not yet sent an acknowledgement to the writer declaring that it has received the sample.

If the low watermark is zero and the unacknowledged sample count decreases to zero, both the low watermark and cache empty events are considered to have taken place. A single callback will be dispatched (assuming the user has requested one) that contains both status changes. The same logic applies when the high watermark is set equal to the maximum number of samples and the cache becomes full.

6.110.2 Member Data Documentation

6.110.2.1 struct DDS_ReliableWriterCacheEventCount DDS_ReliableWriterCacheChangedStatus::empty_reliable_writer_cache [read]

The number of times the reliable writer's cache of unacknowledged samples has become empty.

6.110.2.2 struct DDS_ReliableWriterCacheEventCount DDS_ReliableWriterCacheChangedStatus::full_reliable_writer_cache [read]

The number of times the reliable writer's cache, or send window, of unacknowledged samples has become full.

Applies to writer's cache when the send window is enabled (when both **DDS_RtpsReliableWriterProtocol_t::min_send_window_size** (p. 901) and **DDS_RtpsReliableWriterProtocol_t::max_send_window_size** (p. 901) are **DDS_LENGTH_UNLIMITED** (p. 371)).

Otherwise, applies when the number of unacknowledged samples has reached the send window limit.

6.110.2.3 struct DDS_ReliableWriterCacheEventCount DDS_ReliableWriterCacheChangedStatus::low_watermark_reliable_writer_cache [read]

The number of times the reliable writer's cache of unacknowledged samples has fallen to the low watermark.

A low watermark event will only be considered to have taken place when the number of unacknowledged samples in the writer's cache *decreases* to this value. A sample count that increases to this value will not result in a callback or in a change to the total count of low watermark events.

When the writer's send window is enabled, the low watermark is scaled down, if necessary, to fit within the current send window.

6.110.2.4 struct DDS_ReliableWriterCacheEventCount DDS_ReliableWriterCacheChangedStatus::high_watermark_reliable_writer_cache [read]

The number of times the reliable writer's cache of unacknowledged samples has risen to the high watermark.

A high watermark event will only be considered to have taken place when the number of unacknowledged sampled *increases* to this value. A sample count that was above this value and then decreases back to it will not trigger an event.

When the writer's send window is enabled, the high watermark is scaled down, if necessary, to fit within the current send window.

6.110.2.5 DDS_Long DDS_ReliableWriterCacheChangedStatus::unacknowledged_sample_count

The current number of unacknowledged samples in the writer's cache.

A sample is considered unacknowledged if the writer has failed to receive an acknowledgement from one or more reliable readers matched to it.

6.110.2.6 DDS_Long DDS_ReliableWriterCacheChangedStatus::unacknowledged_sample_count_peak

The highest value that unacknowledged_sample_count has reached until now.

6.111 DDS_ReliableWriterCacheEventCount Struct Reference

*<<**eXtension**>>* *(p. 199)* The number of times the number of unacknowledged samples in the cache of a reliable writer hit a certain well-defined threshold.

Public Attributes

- **DDS_Long total_count**

  The total number of times the event has occurred.

- **DDS_Long total_count_change**

  The incremental number of times the event has occurred since the listener was last invoked or the status read.

6.111.1 Detailed Description

*<<**eXtension**>>* *(p. 199)* The number of times the number of unacknowledged samples in the cache of a reliable writer hit a certain well-defined threshold.

See also:

DDS_ReliableWriterCacheChangedStatus (p. 871)

6.111.2 Member Data Documentation

6.111.2.1 DDS_Long DDS_ReliableWriterCacheEventCount::total_count

The total number of times the event has occurred.

6.111.2.2 DDS_Long DDS_ReliableWriterCacheEventCount::total_count_change

The incremental number of times the event has occurred since the listener was last invoked or the status read.

6.112 DDS_RequestedDeadlineMissedStatus Struct Reference

DDS_REQUESTED_DEADLINE_MISSED_STATUS (p. 323)

Public Attributes

- **DDS_Long total_count**

  *Total cumulative count of the deadlines detected for any instance read by the **DDSDataReader** (p. 1087).*

- **DDS_Long total_count_change**

  The incremental number of deadlines detected since the last time the listener was called or the status was read.

- **DDS_InstanceHandle_t last_instance_handle**

  *Handle to the last instance in the **DDSDataReader** (p. 1087) for which a deadline was detected.*

6.112.1 Detailed Description

DDS_REQUESTED_DEADLINE_MISSED_STATUS (p. 323)

Examples:

HelloWorld_subscriber.cxx.

6.112.2 Member Data Documentation

6.112.2.1 DDS_Long DDS_RequestedDeadlineMissedStatus::total_count

Total cumulative count of the deadlines detected for any instance read by the **DDSDataReader** (p. 1087).

6.112.2.2 DDS_Long DDS_RequestedDeadlineMissedStatus::total_count_change

The incremental number of deadlines detected since the last time the listener was called or the status was read.

6.112.2.3 DDS_InstanceHandle_t DDS_RequestedDeadlineMissedStatus::last_instance_handle

Handle to the last instance in the **DDSDataReader** (p. 1087) for which a deadline was detected.

6.113 DDS_RequestedIncompatibleQosStatus Struct Reference

DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS (p. 323)

Public Attributes

- **DDS_Long total_count**

  Total cumulative count of how many times the concerned ***DDSDataReader*** *(p. 1087) discovered a* ***DDSDataWriter*** *(p. 1113) for the same* ***DDSTopic*** *(p. 1419) with an offered QoS that is incompatible with that requested by the* ***DDSDataReader*** *(p. 1087).*

- **DDS_Long total_count_change**

  The change in `total_count` *since the last time the listener was called or the status was read.*

- **DDS_QosPolicyId_t last_policy_id**

  The PolicyId_t of one of the policies that was found to be incompatible the last time an incompatibility was detected.

- struct **DDS_QosPolicyCountSeq policies**

  A list containing, for each policy, the total number of times that the concerned ***DDSDataReader*** *(p. 1087) discovered a* ***DDSDataWriter*** *(p. 1113) for the same* ***DDSTopic*** *(p. 1419) with an offered QoS that is incompatible with that requested by the* ***DDSDataReader*** *(p. 1087).*

6.113.1 Detailed Description

DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS (p. 323)

See also:

DURABILITY (p. 348)
PRESENTATION (p. 351)
RELIABILITY (p. 362)
OWNERSHIP (p. 355)
LIVELINESS (p. 358)
DEADLINE (p. 353)
LATENCY_BUDGET (p. 354)
DESTINATION_ORDER (p. 365)

Examples:

HelloWorld_subscriber.cxx.

6.113.2 Member Data Documentation

6.113.2.1 DDS_Long DDS_-RequestedIncompatibleQosStatus::total_count

Total cumulative count of how many times the concerned **DDSDataReader** (p. 1087) discovered a **DDSDataWriter** (p. 1113) for the same **DDSTopic** (p. 1419) with an offered QoS that is incompatible with that requested by the **DDSDataReader** (p. 1087).

6.113.2.2 DDS_Long DDS_-RequestedIncompatibleQosStatus::total_count_change

The change in `total_count` since the last time the listener was called or the status was read.

6.113.2.3 DDS_QosPolicyId_t DDS_-RequestedIncompatibleQosStatus::last_policy_id

The PolicyId_t of one of the policies that was found to be incompatible the last time an incompatibility was detected.

6.113.2.4 struct DDS_QosPolicyCountSeq DDS_-RequestedIncompatibleQosStatus::policies [read]

A list containing, for each policy, the total number of times that the concerned **DDSDataReader** (p. 1087) discovered a **DDSDataWriter** (p. 1113) for the same **DDSTopic** (p. 1419) with an offered QoS that is incompatible with that requested by the **DDSDataReader** (p. 1087).

6.114 DDS_ResourceLimitsQosPolicy Struct Reference

Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics.

Public Attributes

- **DDS_Long max_samples**

  Represents the maximum samples the middleware can store for any one ***DDSDataWriter*** *(p. 1113) (or* ***DDSDataReader*** *(p. 1087)).*

- **DDS_Long max_instances**

  Represents the maximum number of instances a ***DDSDataWriter*** *(p. 1113) (or* ***DDSDataReader*** *(p. 1087)) can manage.*

- **DDS_Long max_samples_per_instance**

  Represents the maximum number of samples of any one instance a ***DDSDataWriter*** *(p. 1113) (or* ***DDSDataReader*** *(p. 1087)) can manage.*

- **DDS_Long initial_samples**

  <<**eXtension**>> (p. 199) *Represents the initial samples the middleware will store for any one* ***DDSDataWriter*** *(p. 1113) (or* ***DDSDataReader*** *(p. 1087)).*

- **DDS_Long initial_instances**

  <<**eXtension**>> (p. 199) *Represents the initial number of instances a* ***DDSDataWriter*** *(p. 1113) (or* ***DDSDataReader*** *(p. 1087)) will manage.*

- **DDS_Long instance_hash_buckets**

  <<**eXtension**>> (p. 199) *Number of hash buckets for instances.*

6.114.1 Detailed Description

Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics.

Entity:

DDSTopic (p. 1419), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Status:

DDS_SAMPLE_REJECTED_STATUS (p. 324), **DDS_SampleRejectedStatus** (p. 925)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.114.2 Usage

This policy controls the resources that RTI Connext can use to meet the requirements imposed by the application and other QoS settings.

For the reliability protocol (and **DDS_DurabilityQosPolicy** (p. 614)), this QoS policy determines the actual maximum queue size when the **DDS_HistoryQosPolicy** (p. 758) is set to **DDS_KEEP_ALL_HISTORY_QOS** (p. 368).

In general, this QoS policy is used to limit the amount of system memory that RTI Connext can allocate. For embedded real-time systems and safety-critical systems, pre-determination of maximum memory usage is often required. In addition, dynamic memory allocation could introduce non-deterministic latencies in time-critical paths.

This QoS policy can be set such that an entity does not dynamically allocate any more memory after its initialization phase.

If **DDSDataWriter** (p. 1113) objects are communicating samples faster than they are ultimately taken by the **DDSDataReader** (p. 1087) objects, the middleware will eventually hit against some of the QoS-imposed resource limits. Note that this may occur when just a single **DDSDataReader** (p. 1087) cannot keep up with its corresponding **DDSDataWriter** (p. 1113). The behavior in this case depends on the setting for the **RELIABILITY** (p. 362). If reliability is **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363), then RTI Connext is allowed to drop samples. If the reliability is **DDS_RELIABLE_RELIABILITY_QOS** (p. 363), RTI Connext will block the **DDSDataWriter** (p. 1113) or discard the sample at the **DDSDataReader** (p. 1087) in order not to lose existing samples.

The constant **DDS_LENGTH_UNLIMITED** (p. 371) may be used to indicate the absence of a particular limit. For example setting **DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882) to **DDS_LENGTH_UNLIMITED** (p. 371) will cause RTI Connext not to enforce this particular limit.

If these resource limits are not set sufficiently, under certain circumstances the **DDSDataWriter** (p. 1113) may block on a write() call even though the

DDS_HistoryQosPolicy (p. 758) is **DDS_KEEP_LAST_HISTORY_QOS** (p. 368). To guarantee the writer does not block for **DDS_KEEP_LAST_HISTORY_QOS** (p. 368), make sure the resource limits are set such that:

```
max_samples >= max_instances * max_samples_per_instance
```

See also:

DDS_ReliabilityQosPolicy (p. 865)
DDS_HistoryQosPolicy (p. 758)

6.114.3 Consistency

The setting of **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) must be consistent with **DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882). For these two values to be consistent, it must be true that **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) >= **DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882). As described above, this limit will not be enforced if **DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882) is set to **DDS_LENGTH_UNLIMITED** (p. 371).

The setting of **RESOURCE_LIMITS** (p. 371) `max_samples_per_instance` must be consistent with the **HISTORY** (p. 367) `depth`. For these two QoS to be consistent, it must be true that *depth* <= *max_samples_per_instance*.

See also:

DDS_HistoryQosPolicy (p. 758)

6.114.4 Member Data Documentation

6.114.4.1 DDS_Long DDS_ResourceLimitsQosPolicy::max_samples

Represents the maximum samples the middleware can store for any one **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)).

Specifies the maximum number of data samples a **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)) can manage across all the instances associated with it.

For unkeyed types, this value has to be equal to max_samples_per_instance if max_samples_per_instance is not equal to **DDS_LENGTH_UNLIMITED** (p. 371).

When batching is enabled, the maximum number of data samples a **DDSDataWriter** (p. 1113) can manage will also be limited by **DDS_DataWriterResourceLimitsQosPolicy::max_batches** (p. 563).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 100 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_samples, >= max_samples_per_instance, >= **DDS_DataReaderResourceLimitsQosPolicy::max_samples_per_remote_writer** (p. 524) or >= **DDS_RtpsReliableWriterProtocol_t::heartbeats_per_max_samples** (p. 896)

For **DDS_DataWriterQos** (p. 553) max_samples >= DDS_DataWriterProtocolQosPolicy::rtps_reliable_writer.heartbeats_per_max_samples if batching is disabled.

6.114.4.2 DDS_Long DDS_ResourceLimitsQosPolicy::max_instances

Represents the maximum number of instances a **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)) can manage.

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371), >= initial_instances

6.114.4.3 DDS_Long DDS_ResourceLimitsQosPolicy::max_samples_per_instance

Represents the maximum number of samples of any one instance a **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)) can manage.

For unkeyed types, this value has to be equal to max_samples or **DDS_LENGTH_UNLIMITED** (p. 371).

[default] **DDS_LENGTH_UNLIMITED** (p. 371)

[range] [1, 100 million] or **DDS_LENGTH_UNLIMITED** (p. 371), <= max_samples or **DDS_LENGTH_UNLIMITED** (p. 371), >= **DDS_HistoryQosPolicy::depth** (p. 760)

6.114.4.4 DDS_Long DDS_ResourceLimitsQosPolicy::initial_samples

*<<**eXtension**>>* *(p. 199)* Represents the initial samples the middleware will store for any one **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)).

Specifies the initial number of data samples a **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)) will manage across all the instances associated with it.

[default] 32

[**range**] [1,100 million], <= max_samples

6.114.4.5 DDS_Long DDS_ResourceLimitsQosPolicy::initial_instances

<<***eXtension***>> *(*p. *199)* Represents the initial number of instances a **DDSDataWriter** (p. 1113) (or **DDSDataReader** (p. 1087)) will manage.

[**default**] 32

[**range**] [1,1 million], <= max_instances

6.114.4.6 DDS_Long DDS_ResourceLimitsQosPolicy::instance_hash_buckets

<<***eXtension***>> *(*p. *199)* Number of hash buckets for instances.

The instance hash table facilitates instance lookup. A higher number of buckets decreases instance lookup time but increases the memory usage.

[**default**] 1 [**range**] [1,1 million]

6.115 DDS_RtpsReliableReaderProtocol_t Struct Reference

Qos related to reliable reader protocol defined in RTPS.

Public Attributes

- struct **DDS_Duration_t min_heartbeat_response_delay**

  The minimum delay to respond to a heartbeat.

- struct **DDS_Duration_t max_heartbeat_response_delay**

  The maximum delay to respond to a heartbeat.

- struct **DDS_Duration_t heartbeat_suppression_duration**

  The duration a reader ignores consecutively received heartbeats.

- struct **DDS_Duration_t nack_period**

  The period at which to send NACKs.

- **DDS_Long receive_window_size**

  The number of received out-of-order samples a reader can keep at a time.

- struct **DDS_Duration_t round_trip_time**

  The duration from sending a NACK to receiving a repair of a sample.

- struct **DDS_Duration_t app_ack_period**

  The period at which application-level acknowledgment messages are sent.

- struct **DDS_Duration_t min_app_ack_response_keep_duration**

  Minimum duration for which application-level acknowledgment response data is kept.

- **DDS_Long samples_per_app_ack**

  The minimum number of samples acknowledged by one application-level acknowledgment message.

6.115.1 Detailed Description

Qos related to reliable reader protocol defined in RTPS.

It is used to config reliable reader according to RTPS protocol.

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

QoS:

DDS_DataReaderProtocolQosPolicy (p. 501) **DDS_DiscoveryConfigQosPolicy** (p. 573)

6.115.2 Member Data Documentation

6.115.2.1 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::min_heartbeat_response_delay [read]

The minimum delay to respond to a heartbeat.

When a reliable reader receives a heartbeat from a remote writer and finds out that it needs to send back an ACK/NACK message, the reader can choose to delay a while. This sets the value of the minimum delay.

[**default**] 0 seconds

[**range**] [0, 1 year], <= max_heartbeat_response_delay

6.115.2.2 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::max_heartbeat_response_delay [read]

The maximum delay to respond to a heartbeat.

When a reliable reader receives a heartbeat from a remote writer and finds out that it needs to send back an ACK/NACK message, the reader can choose to delay a while. This sets the value of maximum delay.

[**default**] The default value depends on the container policy:

For **DDS_DiscoveryConfigQosPolicy** (p. 573) : 0 seconds

For **DDS_DataReaderProtocolQosPolicy** (p. 501) : 0.5 seconds

[**range**] [0, 1 year], >= min_heartbeat_response_delay

6.115.2.3 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::heartbeat_suppression_duration [read]

The duration a reader ignores consecutively received heartbeats.

When a reliable reader receives consecutive heartbeats within a short duration that will trigger redundant NACKs, the reader may ignore the latter heartbeat(s). This sets the duration during which additionally received heartbeats are suppressed.

[default] 0.0625 seconds

[range] [0, 1 year],

6.115.2.4 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::nack_period [read]

The period at which to send NACKs.

A reliable reader will send periodic NACKs at this rate when it first matches with a reliable writer. The reader will stop sending NACKs when it has received all available historical data from the writer.

[default] 5 seconds

[range] [1 nanosec, 1 year]

6.115.2.5 DDS_Long DDS_RtpsReliableReaderProtocol_t::receive_window_size

The number of received out-of-order samples a reader can keep at a time.

A reliable reader stores the out-of-order samples it receives until it can present them to the application in-order. The receive window is the maximum number of out-of-order samples that a reliable reader keeps at a given time. When the receive window is full, subsequently received out-of-order samples are dropped.

[default] 256

[range] [>= 1]

6.115.2.6 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::round_trip_time [read]

The duration from sending a NACK to receiving a repair of a sample.

This round-trip time is an estimate of the time starting from when the reader sends a NACK for a specific sample to when it receives that sample. For each sample, the reader will not send a subsequent NACK for it until the round-trip time has passed, thus preventing inefficient redundant requests.

[default] 0 seconds

[**range**] [0 nanosec, 1 year]

6.115.2.7 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::app_ack_period [read]

The period at which application-level acknowledgment messages are sent.

A **DDSDataReader** (p. 1087) sends application-level acknowledgment messages to a **DDSDataWriter** (p. 1113) at this periodic rate, and will continue sending until it receives a message from the **DDSDataWriter** (p. 1113) that it has received and processed the acknowledgment and an AppAckConfirmation has been received by the **DDSDataReader** (p. 1087). Note: application-level acknowledgment messages can also be sent non-periodically, as determined by **DDS_RtpsReliableReaderProtocol_t::samples_per_app_ack** (p. 887).

[**default**] 5 seconds

[**range**] [1 nanosec, 1 year]

6.115.2.8 struct DDS_Duration_t DDS_RtpsReliableReaderProtocol_t::min_app_ack_response_keep_duration [read]

Minimum duration for which application-level acknowledgment response data is kept.

The user-specified response data of an explicit application-level acknowledgment (called by **DDSDataReader::acknowledge_sample** (p. 1095) or **DDSDataReader::acknowledge_all** (p. 1095)) is cached by the **DDSDataReader** (p. 1087) for the purpose of reliably resending the data with the acknowledgment message. After this duration has passed from the time of the first acknowledgment, the response data is dropped from the cache and will not be resent with future acknowledgments for the corresponding sample(s).

[**default**] 0 sec

[**range**] [0 sec, 1 year]

6.115.2.9 DDS_Long DDS_RtpsReliableReaderProtocol_t::samples_per_app_ack

The minimum number of samples acknowledged by one application-level acknowledgment message.

This setting applies only when **DDS_ReliabilityQosPolicy::acknowledgment_kind** (p. 868) = **DDS_APPLICATION_EXPLICIT_**

ACKNOWLEDGMENT_MODE (p. 364) or **DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE** (p. 363)

A **DDSDataReader** (p. 1087) will immediately send an application-level acknowledgment message when it has at least this many samples that have been acknowledged. It will not send an acknowledgment message until it has at least this many samples pending acknowledgment.

For example, calling **DDSDataReader::acknowledge_sample** (p. 1095) this many times consecutively will trigger the sending of an acknowledgment message. Calling **DDSDataReader::acknowledge_all** (p. 1095) may trigger the sending of an acknowledgment message, if at least this many samples are being acknowledged at once.

This is independent of the **DDS_RtpsReliableReaderProtocol_t::app_ack_period** (p. 887), where a **DDSDataReader** (p. 1087) will send acknowledgement messages at the periodic rate regardless.

When this is set to **DDS_LENGTH_UNLIMITED** (p. 371), then acknowledgement messages are sent only periodically, at the rate set by **DDS_RtpsReliableReaderProtocol_t::app_ack_period** (p. 887).

[default] 1

[range] [1, 1000000], or **DDS_LENGTH_UNLIMITED** (p. 371)

6.116 DDS_RtpsReliableWriterProtocol_t Struct Reference

QoS related to the reliable writer protocol defined in RTPS.

Public Attributes

- **DDS_Long low_watermark**

  When the number of unacknowledged samples in the cache of a reliable writer meets or falls below this threshold, the ***DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS*** *(p. 326) is considered to have changed.*

- **DDS_Long high_watermark**

  When the number of unacknowledged samples in the cache of a reliable writer meets or exceeds this threshold, the ***DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS*** *(p. 326) is considered to have changed.*

- struct **DDS_Duration_t heartbeat_period**

  The period at which to send heartbeats.

- struct **DDS_Duration_t fast_heartbeat_period**

  An alternative heartbeat period used when a reliable writer needs to flush its unacknowledged samples more quickly.

- struct **DDS_Duration_t late_joiner_heartbeat_period**

  An alternative heartbeat period used when a reliable reader joins late and needs to be caught up on cached samples of a reliable writer more quickly than the normal heartbeat rate.

- struct **DDS_Duration_t virtual_heartbeat_period**

  The period at which to send virtual heartbeats. Virtual heartbeats inform the reliable reader about the range of samples currently present, for each virtual GUID, in the reliable writer's queue.

- **DDS_Long samples_per_virtual_heartbeat**

  The number of samples that a reliable writer has to publish before sending a virtual heartbeat.

- **DDS_Long max_heartbeat_retries**

  The maximum number of periodic *heartbeat retries before marking a remote reader as inactive.*

- **DDS_Boolean inactivate_nonprogressing_readers**

Whether to treat remote readers as inactive when their NACKs do not progress.

^ **DDS_Long heartbeats_per_max_samples**

The number of heartbeats per send queue.

^ struct **DDS_Duration_t min_nack_response_delay**

The minimum delay to respond to a NACK.

^ struct **DDS_Duration_t max_nack_response_delay**

The maximum delay to respond to a nack.

^ struct **DDS_Duration_t nack_suppression_duration**

The duration for ignoring consecutive NACKs that may trigger redundant repairs.

^ **DDS_Long max_bytes_per_nack_response**

The maximum total message size when resending dropped samples.

^ struct **DDS_Duration_t disable_positive_acks_min_sample_keep_duration**

The minimum duration a sample is queued for ACK-disabled readers.

^ struct **DDS_Duration_t disable_positive_acks_max_sample_keep_duration**

The maximum duration a sample is queued for ACK-disabled readers.

^ **DDS_Boolean disable_positive_acks_enable_adaptive_sample_keep_duration**

Enables dynamic adjustment of sample keep duration in response to congestion.

^ **DDS_Long disable_positive_acks_decrease_sample_keep_duration_factor**

Controls rate of contraction of dynamic sample keep duration.

^ **DDS_Long disable_positive_acks_increase_sample_keep_duration_factor**

Controls rate of growth of dynamic sample keep duration.

^ **DDS_Long min_send_window_size**

Minimum size of send window of unacknowledged samples.

ˆ **DDS_Long max_send_window_size**

Maximum size of send window of unacknowledged samples.

ˆ struct **DDS_Duration_t send_window_update_period**

Period in which send window may be dynamically changed.

ˆ **DDS_Long send_window_increase_factor**

Increases send window size by this percentage when reacting dynamically to network conditions.

ˆ **DDS_Long send_window_decrease_factor**

Decreases send window size by this percentage when reacting dynamically to network conditions.

ˆ **DDS_Boolean enable_multicast_periodic_heartbeat**

Whether periodic heartbeat messages are sent over multicast.

ˆ **DDS_Long multicast_resend_threshold**

The minimum number of requesting readers needed to trigger a multicast resend.

6.116.1 Detailed Description

QoS related to the reliable writer protocol defined in RTPS.

It is used to configure a reliable writer according to RTPS protocol.

The reliability protocol settings are applied to batches instead of individual data samples when batching is enabled.

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

QoS:

DDS_DataWriterProtocolQosPolicy (p. 535) **DDS_DiscoveryConfigQosPolicy** (p. 573)

6.116.2 Member Data Documentation

6.116.2.1 DDS_Long DDS_RtpsReliableWriterProtocol_t::low_watermark

When the number of unacknowledged samples in the cache of a reliable writer meets or falls below this threshold, the **DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS** (p. 326) is considered to have changed.

This value is measured in units of samples, except with batching configurations in non-MultiChannel DataWriters where it is measured in units of batches.

The value must be greater than or equal to zero and strictly less than high_watermark.

The high and low watermarks are used for switching between the regular and fast heartbeat rates (**DDS_RtpsReliableWriterProtocol_t::heartbeat_period** (p. 893) and **DDS_RtpsReliableWriterProtocol_t::fast_heartbeat_period** (p. 894), respectively). When the number of unacknowledged samples in the queue of a reliable **DDSDataWriter** (p. 1113) meets or exceeds high_watermark, the **DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS** (p. 326) is changed, and the DataWriter will start heartbeating at fast_heartbeat_rate. When the number of samples meets or falls below low_watermark, **DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS** (p. 326) is changed, and the heartbeat rate will return to the "normal" rate (heartbeat_rate).

[default] 0

[range] [0, 100 million], < high_watermark

See also:

Multi-channel DataWriters (p. 122) for additional details on reliability with MultChannel DataWriters.

6.116.2.2 DDS_Long DDS_RtpsReliableWriterProtocol_t::high_watermark

When the number of unacknowledged samples in the cache of a reliable writer meets or exceeds this threshold, the **DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS** (p. 326) is considered to have changed.

This value is measured in units of samples, except with batching configurations in non-MultiChannel DataWriters where it is measured in units of batches.

The value must be strictly greater than low_watermark and less than or equal to a maximum that depends on the container QoS policy:

In **DDS_DomainParticipantQos::discovery_config** (p. 591):

For **DDS_DiscoveryConfigQosPolicy::publication_writer** (p. 580)

high_watermark<=DDS_DomainParticipantQos::resource_limits.local_writer_allocation.max_count

For **DDS_DiscoveryConfigQosPolicy::subscription_writer** (p. 580)

high_watermark<=DDS_DomainParticipantQos::resource_limits.local_reader_allocation.max_count

In **DDS_DataWriterQos::protocol** (p. 558):

For **DDS_DataWriterProtocolQosPolicy::rtps_reliable_writer** (p. 539),

high_watermark<=**DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) if batching is disabled or the DataWriter is a MultiChannel DataWriter. Otherwise,

high_watermark<=**DDS_DataWriterResourceLimitsQosPolicy::max_batches** (p. 563)

[**default**] 1

[**range**] [1, 100 million] or **DDS_LENGTH_UNLIMITED** (p. 371), > low_watermark <= *maximum* which depends on the container policy

See also:

Multi-channel DataWriters (p. 122) for additional details on reliability with MultChannel DataWriters.

6.116.2.3 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::heartbeat_period [read]

The period at which to send heartbeats.

A reliable writer will send periodic heartbeats at this rate.

[**default**] 3 seconds

[**range**] [1 nanosec,1 year], >= **DDS_RtpsReliableWriterProtocol_t::fast_heartbeat_period** (p. 894), >= **DDS_RtpsReliableWriterProtocol_t::late_joiner_heartbeat_period** (p. 894)

6.116.2.4 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::fast_heartbeat_period [read]

An alternative heartbeat period used when a reliable writer needs to flush its unacknowledged samples more quickly.

This heartbeat period will be used when the number of unacknowledged samples in the cache of a reliable writer meets or exceeds the writer's high watermark and has not subsequently dropped to the low watermark. The normal period will be used at all other times.

This period must not be slower (i.e. must be of the same or shorter duration) than the normal heartbeat period.

[default] 3 seconds

[range] [1 nanosec,1 year], <= **DDS_RtpsReliableWriterProtocol_t::heartbeat_period** (p. 893)

6.116.2.5 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::late_joiner_heartbeat_period [read]

An alternative heartbeat period used when a reliable reader joins late and needs to be caught up on cached samples of a reliable writer more quickly than the normal heartbeat rate.

This heartbeat period will be used when a reliable reader joins after a reliable writer with non-volatile durability has begun publishing samples. Once the reliable reader has received all cached samples, it will be serviced at the same rate as other reliable readers.

This period must not be slower (i.e. must be of the same or shorter duration) than the normal heartbeat period.

[default] 3 seconds

[range] [1 nanosec,1 year], <= **DDS_RtpsReliableWriterProtocol_t::heartbeat_period** (p. 893)

6.116.2.6 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::virtual_heartbeat_period [read]

The period at which to send virtual heartbeats. Virtual heartbeats inform the reliable reader about the range of samples currently present, for each virtual GUID, in the reliable writer's queue.

A reliable writer will send periodic virtual heartbeats at this rate.

[**default**] **DDS_DURATION_AUTO** (p. 305). If **DDS_PresentationQosPolicy::access_scope** (p. 826) is set to **DDS_GROUP_PRESENTATION_QOS** (p. 352), this value is set to **DDS_RtpsReliableWriterProtocol_t::heartbeat_period** (p. 893). Otherwise, the value is set to **DDS_DURATION_INFINITE** (p. 305).

[**range**] > 1 nanosec, **DDS_DURATION_INFINITE** (p. 305), or **DDS_DURATION_AUTO** (p. 305)

6.116.2.7 DDS_Long DDS_RtpsReliableWriterProtocol_t::samples_per_virtual_heartbeat

The number of samples that a reliable writer has to publish before sending a virtual heartbeat.

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

[**range**] [1,1000000], **DDS_LENGTH_UNLIMITED** (p. 371)

6.116.2.8 DDS_Long DDS_RtpsReliableWriterProtocol_t::max_heartbeat_retries

The maximum number of *periodic* heartbeat retries before marking a remote reader as inactive.

When a remote reader has not acked all the samples the reliable writer has in its queue, and max_heartbeat_retries number of periodic heartbeats has been sent without receiving any ack/nack back, the remote reader will be marked as inactive (not alive) and be ignored until it resumes sending ack/nack.

Note that piggyback heartbeats do NOT count towards this value.

[**default**] 10

[**range**] [1, 1 million] or **DDS_LENGTH_UNLIMITED** (p. 371)

6.116.2.9 DDS_Boolean DDS_RtpsReliableWriterProtocol_t::inactivate_nonprogressing_readers

Whether to treat remote readers as inactive when their NACKs do not progress.

Nominally, a remote reader is marked inactive when a successive number of periodic heartbeats equal or greater than **DDS_RtpsReliableWriterProtocol_t::max_heartbeat_retries** (p. 895) have been sent without receiving any ack/nacks back.

By setting this **DDS_BOOLEAN_TRUE** (p. 298), it changes the conditions

of inactivating a remote reader: a reader will be considered inactive when it either does not send any ack/nacks or keeps sending non-progressing nacks for **DDS_RtpsReliableWriterProtocol_t::max_heartbeat_retries** (p. 895) number of heartbeat periods, where a non-progressing nack is one whose oldest sample requested has not advanced from the oldest sample requested of the previous nack.

[default] **DDS_BOOLEAN_FALSE** (p. 299)

6.116.2.10 DDS_Long DDS_RtpsReliableWriterProtocol_t::heartbeats_per_max_samples

The number of heartbeats per send queue.

If batching is disabled or the DataWriter is a MultiChannel DataWriter: a piggyback heartbeat will be sent every [**DDS_ResourceLimitsQosPolicy::max_samples** (p. 881)/heartbeats_per_max_samples] number of samples.

Otherwise: a piggyback heartbeat will be sent every [**DDS_DataWriterResourceLimitsQosPolicy::max_batches** (p. 563)/heartbeats_per_max_samples] number of batches.

If set to zero, no piggyback heartbeat will be sent. If maximum is **DDS_LENGTH_UNLIMITED** (p. 371), 100 million is assumed as the maximum value in the calculation.

[default] 8

[range] [0, 100 million]

- For **DDS_DiscoveryConfigQosPolicy::publication_writer** (p. 580):

  heartbeats_per_max_samples<= DDS_DomainParticipantQos::resource_limits.local_writer_allocation.max_count

- For **DDS_DiscoveryConfigQosPolicy::subscription_writer** (p. 580):

  heartbeats_per_max_samples<= DDS_DomainParticipantQos::resource_limits.local_reader_allocation.max_count

- For **DDS_DataWriterProtocolQosPolicy::rtps_reliable_writer** (p. 539):

  heartbeats_per_max_samples<= **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) if batching is disabled or the DataWriter is a MultiChannel DataWriter. Otherwise:

  heartbeats_per_max_samples<= **DDS_DataWriterResourceLimitsQosPolicy::max_batches** (p. 563).

6.116.2.11 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::min_nack_response_delay [read]

The minimum delay to respond to a NACK.

When a reliable writer receives a NACK from a remote reader, the writer can choose to delay a while before it sends repair samples or a heartbeat. This sets the value of the minimum delay.

[**default**] 0 seconds

[**range**] [0,1 day], <= max_nack_response_delay

6.116.2.12 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::max_nack_response_delay [read]

The maximum delay to respond to a nack.

This set the value of maximum delay between receiving a NACK and sending repair samples or a heartbeat.

[**default**] The default value depends on the container policy:

For **DDS_DiscoveryConfigQosPolicy** (p. 573) : 0 seconds

For **DDS_DataWriterProtocolQosPolicy** (p. 535) : 0.2 seconds

[**range**] [0,1 day], >= min_nack_response_delay

6.116.2.13 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::nack_suppression_duration [read]

The duration for ignoring consecutive NACKs that may trigger redundant repairs.

A reliable writer may receive consecutive NACKs within a short duration from a remote reader that will trigger the sending of redundant repair messages.

This specifies the duration during which consecutive NACKs are ignored to prevent redundant repairs from being sent.

[**default**] 0 seconds

[**range**] [0,1 day],

6.116.2.14 DDS_Long DDS_RtpsReliableWriterProtocol_t::max_bytes_per_nack_response

The maximum total message size when resending dropped samples.

As part of the reliable communication protocol, data writers send heartbeat (HB) messages to their data readers. Each HB message contains the sequence number of the most recent sample sent by the data writer.

In response, a data reader sends an acknowledgement (ACK) message, indicating what sequence numbers it did not receive, if any. If the data reader is missing some samples, the data writer will send them again.

max_bytes_per_nack_response determines the maximum size of the message sent by the data writer in response to an ACK. This message may contain multiple samples.

If max_bytes_per_nack_response is larger than the maximum message size supported by the underlying transport, RTI Connext will send multiple messages. If the total size of all samples that need to be resent is larger than max_bytes_per_nack_response, the remaining samples will be resent the next time an ACK arrives.

[default] 131072

[range] [0, 1 GB]

6.116.2.15 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_min_sample_keep_duration [read]

The minimum duration a sample is queued for ACK-disabled readers.

When positive ACKs are disabled for a data writer (**DDS_DataWriterProtocolQosPolicy::disable_positive_acks** (p. 537) = **DDS_BOOLEAN_TRUE** (p. 298)) or a data reader (**DDS_DataReaderProtocolQosPolicy::disable_positive_acks** (p. 503) = **DDS_BOOLEAN_TRUE** (p. 298)), a sample is available from the data writer's queue for at least this duration, after which the sample may be considered to be acknowledged.

[default] 1 millisecond

[range] [0,1 year], <= **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_max_sample_keep_duration** (p. 899)

6.116.2.16 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_max_sample_keep_duration [read]

The maximum duration a sample is queued for ACK-disabled readers.

When positive ACKs are disabled for a data writer (**DDS_DataWriterProtocolQosPolicy::disable_positive_acks** (p. 537) = **DDS_BOOLEAN_TRUE** (p. 298)) or a data reader (**DDS_DataReaderProtocolQosPolicy::disable_positive_acks** (p. 503) = **DDS_BOOLEAN_TRUE** (p. 298)), a sample is available from the data writer's queue for at most this duration, after which the sample is considered to be acknowledged.

[**default**] 1 second

[**range**] [0,1 year], >= **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_min_sample_keep_duration** (p. 898)

6.116.2.17 DDS_Boolean DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_enable_adaptive_sample_keep_duration

Enables dynamic adjustment of sample keep duration in response to congestion.

For dynamic networks where a static minimum sample keep duration may not provide sufficient performance or reliability, setting **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_enable_adaptive_sample_keep_duration** (p. 899) = **DDS_BOOLEAN_TRUE** (p. 298), enables the sample keep duration to be dynamically adjusted to adapt to network conditions. The keep duration changes according to the detected level of congestion, which is determined to be proportional to the rate of NACKs received. An adaptive algorithm automatically controls the keep duration to optimize throughput and reliability.

To relieve high congestion, the keep duration is increased to effectively decrease the send rate; this lengthening of the keep duration is controlled by **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_increase_sample_keep_duration_factor** (p. 900). Alternatively, when congestion is low, the keep duration is decreased to effectively increase send rate; this shortening of the keep duration is controlled by **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_decrease_sample_keep_duration_factor** (p. 900).

The lower and upper bounds of the dynamic sample keep duration are set by **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_min_sample_keep_duration** (p. 898) and **DDS_RtpsReliableWriterProtocol_**

t::disable_positive_acks_max_sample_keep_duration (p. 899), respectively.

When **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_enable_adaptive_sample_keep_duration** (p. 899) = **DDS_BOOLEAN_FALSE** (p. 299), the sample keep duration is set to **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_min_sample_keep_duration** (p. 898) .

[**default**] **DDS_BOOLEAN_TRUE** (p. 298)

6.116.2.18 DDS_Long DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_decrease_sample_keep_duration_factor

Controls rate of contraction of dynamic sample keep duration.

Used when **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_enable_adaptive_sample_keep_duration** (p. 899) = **DDS_BOOLEAN_TRUE** (p. 298).

When the adaptive algorithm determines that the keep duration should be decreased, this factor (a percentage) is multiplied with the current keep duration to get the new shorter keep duration. For example, if the current keep duration is 20 milliseconds, using the default factor of 95% would result in a new keep duration of 19 milliseconds.

[**default**] 95

[**range**] <= 100

6.116.2.19 DDS_Long DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_increase_sample_keep_duration_factor

Controls rate of growth of dynamic sample keep duration.

Used when **DDS_RtpsReliableWriterProtocol_t::disable_positive_acks_enable_adaptive_sample_keep_duration** (p. 899) = **DDS_BOOLEAN_TRUE** (p. 298).

When the adaptive algorithm determines that the keep duration should be increased, this factor (a percentage) is multiplied with the current keep duration to get the new longer keep duration. For example, if the current keep duration is 20 milliseconds, using the default factor of 150% would result in a new keep duration of 30 milliseconds.

[**default**] 150

[**range**] >= 100

6.116.2.20 DDS_Long DDS_RtpsReliableWriterProtocol_t::min_send_window_size

Minimum size of send window of unacknowledged samples.

A **DDSDataWriter** (p. 1113) has a limit on the number of unacknowledged samples in-flight at a time. This send window can be configured to have a minimum size (this field) and a maximum size (max_send_window_size). The send window can dynamically change, between the min and max sizes, to throttle the effective send rate in response to changing network congestion, as measured by negative acknowledgements received.

When both min_send_window_size and max_send_window_size are **DDS_LENGTH_UNLIMITED** (p. 371), then **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) serves as the effective send window limit.

When **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) is less than max_send_window_size, then it serves as the effective max send window. If it is less than min_send_window_size, then effectively both min and max send window sizes are equal to max samples. In addition, the low and high watermarks are scaled down linearly to stay within the send window size, and the full reliable queue status is set when the send window is full.

[**default**] **DDS_LENGTH_UNLIMITED** (p. 371)

[**range**] > 0, <= max_send_window_size, or **DDS_LENGTH_UNLIMITED** (p. 371)

See also:

DDS_RtpsReliableWriterProtocol_t::max_send_window_size (p. 901)
DDS_RtpsReliableWriterProtocol_t::low_watermark (p. 892)
DDS_RtpsReliableWriterProtocol_t::high_watermark (p. 892)
DDS_ReliableWriterCacheChangedStatus::full_reliable_writer_cache (p. 872)

6.116.2.21 DDS_Long DDS_RtpsReliableWriterProtocol_t::max_send_window_size

Maximum size of send window of unacknowledged samples.

A **DDSDataWriter** (p. 1113) has a limit on the number of unacknowledged samples in-flight at a time. This send window can be configured to have a minimum size (min_send_window_size) and a maximum size (this field). The send window can dynamically change, between the min and max sizes, to throttle

the effective send rate in response to changing network congestion, as measured by negative acknowledgements received.

When both min_send_window_size and max_send_window_size are **DDS_LENGTH_UNLIMITED** (p. 371), then **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) serves as the effective send window limit. When **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) is less than max_send_window_size, then it serves as the effective max send window. If it is also less than min_send_window_size, then effectively both min and max send window sizes are equal to max samples. In addition, the low and high watermarks are scaled down linearly to stay within the send window size, and the full reliable queue status is set when the send window is full.

[default] DDS_LENGTH_UNLIMITED (p. 371)

[range] > 0, >= min_send_window_size, or **DDS_LENGTH_UNLIMITED** (p. 371)

See also:

DDS_RtpsReliableWriterProtocol_t::min_send_window_size (p. 901)
DDS_RtpsReliableWriterProtocol_t::low_watermark (p. 892)
DDS_RtpsReliableWriterProtocol_t::high_watermark (p. 892)
DDS_ReliableWriterCacheChangedStatus::full_reliable_writer_cache (p. 872)

6.116.2.22 struct DDS_Duration_t DDS_RtpsReliableWriterProtocol_t::send_window_update_period [read]

Period in which send window may be dynamically changed.

The **DDSDataWriter** (p. 1113)'s send window will dynamically change, between the min and max send window sizes, to throttle the effective send rate in response to changing network congestion, as measured by negative acknowledgements received.

The change in send window size happens at this update period, whereupon the send window is either increased or decreased in size according to the increase or decrease factors, respectively.

[default] 3 seconds

[range] > [0,1 year]

See also:

DDS_RtpsReliableWriterProtocol_t::send_window_increase_

factor (p. 903), **DDS_RtpsReliableWriterProtocol_t::send_window_decrease_factor** (p. 903)

6.116.2.23 DDS_Long DDS_RtpsReliableWriterProtocol_t::send_window_increase_factor

Increases send window size by this percentage when reacting dynamically to network conditions.

The **DDSDataWriter** (p. 1113)'s send window will dynamically change, between the min and max send window sizes, to throttle the effective send rate in response to changing network congestion, as measured by negative acknowledgements received.

After an update period during which no negative acknowledgements were received, the send window will be increased by this factor. The factor is treated as a percentage, where a factor of 150 would increase the send window by 150%. The increased send window size will not exceed the max_send_window_size.

[**default**] 105

[**range**] > 100

See also:

DDS_RtpsReliableWriterProtocol_t::send_window_update_period (p. 902), **DDS_RtpsReliableWriterProtocol_t::send_window_decrease_factor** (p. 903)

6.116.2.24 DDS_Long DDS_RtpsReliableWriterProtocol_t::send_window_decrease_factor

Decreases send window size by this percentage when reacting dynamically to network conditions.

The **DDSDataWriter** (p. 1113)'s send window will dynamically change, between the min and max send window sizes, to throttle the effective send rate in response to changing network congestion, as measured by negative acknowledgements received.

When increased network congestion causes a negative acknowledgement to be received by a writer, the send window will be decreased by this factor to throttle the effective send rate. The factor is treated as a percentage, where a factor of 80 would decrease the send window to 80% of its previous size. The decreased send window size will not be less than the min_send_window_size.

[**default**] 70

[range] [0, 100]

See also:

DDS_RtpsReliableWriterProtocol_t::send_window_update_period (p. 902), **DDS_RtpsReliableWriterProtocol_t::send_window_increase_factor** (p. 903)

6.116.2.25 DDS_Boolean DDS_RtpsReliableWriterProtocol_t::enable_multicast_periodic_heartbeat

Whether periodic heartbeat messages are sent over multicast.

When enabled, if a reader has a multicast destination, then the writer will send its periodic HEARTBEAT messages to that destination. Otherwise, if not enabled or the reader does not have a multicast destination, the writer will send its periodic HEARTBEATs over unicast.

[default] **DDS_BOOLEAN_FALSE** (p. 299)

6.116.2.26 DDS_Long DDS_RtpsReliableWriterProtocol_t::multicast_resend_threshold

The minimum number of requesting readers needed to trigger a multicast resend.

Given readers with multicast destinations, when a reader NACKs for samples to be resent, the writer can either resend them over unicast or multicast. In order for the writer to resend over multicast, this threshold is the minimum number of readers of the same multicast group that the writer must receive NACKs from within a single response-delay. This allows the writer to coalesce near-simultaneous unicast resends into a multicast resend. Note that a threshold of 1 means that all resends will be sent over multicast, if available.

[default] 2

[range] [>= 1]

6.117 DDS_RtpsWellKnownPorts_t Struct Reference

RTPS well-known port mapping configuration.

Public Attributes

- **DDS_Long port_base**
  The base port offset.
- **DDS_Long domain_id_gain**
  Tunable domain gain parameter.
- **DDS_Long participant_id_gain**
  Tunable participant gain parameter.
- **DDS_Long builtin_multicast_port_offset**
  Additional offset for **metatraffic** *multicast port.*
- **DDS_Long builtin_unicast_port_offset**
  Additional offset for **metatraffic** *unicast port.*
- **DDS_Long user_multicast_port_offset**
  Additional offset for **usertraffic** *multicast port.*
- **DDS_Long user_unicast_port_offset**
  Additional offset for **usertraffic** *unicast port.*

6.117.1 Detailed Description

RTPS well-known port mapping configuration.

RTI Connext uses the RTPS wire protocol. The discovery protocols defined by RTPS rely on well-known ports to initiate discovery. These well-known ports define the multicast and unicast ports on which a Participant will listen for discovery **metatraffic** from other Participants. The discovery metatraffic contains all the information required to establish the presence of remote DDS entities in the network.

The well-known ports are defined by RTPS in terms of port mapping expressions with several tunable parameters, which allow you to customize what network ports are used by RTI Connext. These parameters are exposed in **DDS_RtpsWellKnownPorts_t** (p. 905). In order for all Participants in a system to

correctly discover each other, it is important that they all use the same port mapping expressions.

The actual port mapping expressions, as defined by the RTPS specification, can be found below. In addition to the parameters listed in **DDS_RtpsWellKnownPorts_t** (p. 905), the port numbers depend on:

- `domain_id`, as specified in **DDSDomainParticipantFactory::create_participant** (p. 1233)

- `participant_id`, as specified using **DDS_WireProtocolQosPolicy::participant_id** (p. 1063)

The `domain_id` parameter ensures no port conflicts exist between Participants belonging to different domains. This also means that discovery metatraffic in one domain is not visible to Participants in a different domain. The `participant_id` parameter ensures that unique unicast port numbers are assigned to Participants belonging to the same domain on a given host.

The *metatraffic_unicast_port* is used to exchange discovery metatraffic using unicast.

```
metatraffic_unicast_port = port_base + (domain_id_gain * domain_id) + (participant_id_gain * participant_id
```

The *metatraffic_multicast_port* is used to exchange discovery metatraffic using multicast. The corresponding multicast group addresses are specified via **DDS_DiscoveryQosPolicy::multicast_receive_addresses** (p. 584) on a **DDSDomainParticipant** (p. 1139) entity.

```
metatraffic_multicast_port = port_base + (domain_id_gain * domain_id) + builtin_multicast_port_offset
```

RTPS also defines the *default* multicast and unicast ports on which DataReaders and DataWriters receive **usertraffic**. These default ports can be overridden using the **DDS_DataReaderQos::multicast** (p. 520), **DDS_DataReaderQos::unicast** (p. 519), or by the **DDS_DataWriterQos::unicast** (p. 558) QoS policies.

The *usertraffic_unicast_port* is used to exchange user data using unicast.

```
usertraffic_unicast_port = port_base + (domain_id_gain * domain_id) + (participant_id_gain * participant_id
```

The *usertraffic_multicast_port* is used to exchange user data using multicast. The corresponding multicast group addresses can be configured using **DDS_TransportMulticastQosPolicy** (p. 978).

```
usertraffic_multicast_port = port_base + (domain_id_gain * domain_id) + user_multicast_port_offset
```

By default, the port mapping parameters are configured to compliant with OMG's DDS Interoperability Wire Protocol (see also **DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS** (p. 405)).

The OMG's DDS Interoperability Wire Protocol compliant port mapping parameters are *not* backwards compatible with previous versions of the RTI Connext middleware.

When modifying the port mapping parameters, care must be taken to avoid port aliasing. This would result in undefined discovery behavior. The chosen parameter values will also determine the maximum possible number of domains in the system and the maximum number of participants per domain. Additionally, any resulting mapped port number must be within the range imposed by the underlying transport. For example, for UDPv4, this range typically equals [1024 - 65535].

QoS:

DDS_WireProtocolQosPolicy (p. 1059)

6.117.2 Member Data Documentation

6.117.2.1 DDS_Long DDS_RtpsWellKnownPorts_t::port_base

The base port offset.

All mapped well-known ports are offset by this value.

[**default**] 7400

[**range**] [>= 1], but resulting ports must be within the range imposed by the underlying transport.

6.117.2.2 DDS_Long DDS_RtpsWellKnownPorts_t::domain_id_gain

Tunable domain gain parameter.

Multiplier of the `domain_id`. Together with `participant_id_gain`, it determines the highest `domain_id` and `participant_id` allowed on this network.

In general, there are two ways to setup `domain_id_gain` and `participant_id_gain` parameters.

If `domain_id_gain` > `participant_id_gain`, it results in a port mapping layout where all **DDSDomainParticipant** (p. 1139) instances within a single domain occupy a consecutive range of `domain_id_gain` ports. Precisely, all ports occupied by the domain fall within:

```
(port_base + (domain_id_gain * domain_id))
```

and:

```
(port_base + (domain_id_gain * (domain_id + 1)) - 1)
```

Under such a case, the highest `domain_id` is limited only by the underlying transport's maximum port. The highest `participant_id`, however, must satisfy:

```
max_participant_id < (domain_id_gain / participant_id_gain)
```

On the contrary, if `domain_id_gain` <= `participant_id_gain`, it results in a port mapping layout where a given domain's **DDSDomainParticipant** (p. 1139) instances occupy ports spanned across the entire valid port range allowed by the underlying transport. For instance, it results in the following potential mapping:

Mapped Port	Domain Id	Participant ID
higher port number	Domain Id = 1	Participant ID = 2
	Domain Id = 0	Participant ID = 2
	Domain Id = 1	Participant ID = 1
	Domain Id = 0	Participant ID = 1
	Domain Id = 1	Participant ID = 0
lower port number	Domain Id = 0	Participant ID = 0

Under this case, the highest `participant_id` is limited only by the underlying transport's maximum port. The highest `domain_id`, however, must satisfy:

```
max_domain_id < (participant_id_gain / domain_id_gain)
```

Additionally, `domain_id_gain` also determines the range of the port-specific offsets.

```
domain_id_gain > abs(builtin_multicast_port_offset - user_multicast_port_offset)
```

```
domain_id_gain > abs(builtin_unicast_port_offset - user_unicast_port_offset)
```

Violating this may result in port aliasing and undefined discovery behavior.

[default] 250

[range] [> 0], but resulting ports must be within the range imposed by the underlying transport.

6.117.2.3 DDS_Long DDS_RtpsWellKnownPorts_t::participant_id_gain

Tunable participant gain parameter.

Multiplier of the participant_id. See **DDS_RtpsWellKnownPorts_t::domain_id_gain** (p. 907) for its implications on the highest domain_id and participant_id allowed on this network.

Additionally, participant_id_gain also determines the range of builtin_unicast_port_offset and user_unicast_port_offset.

```
participant_id_gain > abs(builtin_unicast_port_offset - user_unicast_port_offset)
```

[**default**] 2

[**range**] [> 0], but resulting ports must be within the range imposed by the underlying transport.

6.117.2.4 DDS_Long DDS_RtpsWellKnownPorts_t::builtin_multicast_port_offset

Additional offset for **metatraffic** multicast port.

It must be unique from other port-specific offsets.

[**default**] 0

[**range**] [>= 0], but resulting ports must be within the range imposed by the underlying transport.

6.117.2.5 DDS_Long DDS_RtpsWellKnownPorts_t::builtin_unicast_port_offset

Additional offset for **metatraffic** unicast port.

It must be unique from other port-specific offsets.

[**default**] 10

[**range**] [>= 0], but resulting ports must be within the range imposed by the underlying transport.

6.117.2.6 DDS_Long DDS_RtpsWellKnownPorts_t::user_multicast_port_offset

Additional offset for **usertraffic** multicast port.

It must be unique from other port-specific offsets.

[**default**] 1

[**range**] [>= 0], but resulting ports must be within the range imposed by the underlying transport.

6.117.2.7 DDS_Long DDS_RtpsWellKnownPorts_t::user_unicast_port_offset

Additional offset for **usertraffic** unicast port.

It must be unique from other port-specific offsets.

[**default**] 11

[**range**] [>= 0], but resulting ports must be within the range imposed by the underlying transport.

6.118 DDS_SampleIdentity_t Struct Reference

Type definition for an Sample Identity.

Public Attributes

- struct **DDS_GUID_t writer_guid**

  16-byte identifier identifying the virtual GUID.

- struct **DDS_SequenceNumber_t sequence_number**

  monotonically increasing 64-bit integer that identifies the sample in the data source.

6.118.1 Detailed Description

Type definition for an Sample Identity.

A SampleIdentity defines a pair (Virtual Writer GUID, Sequence Number) that uniquely identifies a sample within a DDS domain and a Topic.

6.119 DDS_SampleInfo Struct Reference

Information that accompanies each sample that is `read` or `taken`.

Public Attributes

- **DDS_SampleStateKind sample_state**

  The sample state of the sample.

- **DDS_ViewStateKind view_state**

  The view state of the instance.

- **DDS_InstanceStateKind instance_state**

  The instance state of the instance.

- struct **DDS_Time_t source_timestamp**

  The timestamp when the sample was written by a DataWriter.

- **DDS_InstanceHandle_t instance_handle**

  Identifies locally the corresponding instance.

- **DDS_InstanceHandle_t publication_handle**

  Identifies locally the DataWriter that modified the instance.

- **DDS_Long disposed_generation_count**

  The disposed generation count of the instance at the time of sample reception.

- **DDS_Long no_writers_generation_count**

  The no writers generation count of the instance at the time of sample reception.

- **DDS_Long sample_rank**

  The sample rank of the sample.

- **DDS_Long generation_rank**

  The generation rank of the sample.

- **DDS_Long absolute_generation_rank**

  The absolute generation rank of the sample.

- **DDS_Boolean valid_data**

Indicates whether the `DataSample` *contains data or else it is only used to communicate a change in the* `instance_state` *of the instance.*

- struct **DDS_Time_t reception_timestamp**

  <<**eXtension**>> (p. 199) *The timestamp when the sample was committed by a DataReader.*

- struct **DDS_SequenceNumber_t publication_sequence_number**

  <<**eXtension**>> (p. 199) *The publication sequence number.*

- struct **DDS_SequenceNumber_t reception_sequence_number**

  <<**eXtension**>> (p. 199) *The reception sequence number when sample was committed by a DataReader*

- struct **DDS_GUID_t original_publication_virtual_guid**

  <<**eXtension**>> (p. 199) *The original publication virtual GUID.*

- struct **DDS_SequenceNumber_t original_publication_virtual_sequence_number**

  <<**eXtension**>> (p. 199) *The original publication virtual sequence number.*

- struct **DDS_GUID_t related_original_publication_virtual_guid**

  <<**eXtension**>> (p. 199) *The original publication virtual GUID of a related sample*

- struct **DDS_SequenceNumber_t related_original_publication_virtual_sequence_number**

  <<**eXtension**>> (p. 199) *The original publication virtual sequence number of a related sample*

6.119.1 Detailed Description

Information that accompanies each sample that is `read` or `taken`.

6.119.2 Interpretation of the SampleInfo

The **DDS_SampleInfo** (p. 912) contains information pertaining to the associated `Data` instance sample including:

- the `sample_state` of the `Data` value (i.e., if it has already been read or not)

- the `view_state` of the related instance (i.e., if the instance is new or not)
- the `instance_state` of the related instance (i.e., if the instance is alive or not)
- the `valid_data` flag. This flag indicates whether there is data associated with the sample. Some samples do not contain data indicating only a change on the `instance_state` of the corresponding instance.
- The values of `disposed_generation_count` and `no_writers_generation_count` for the related instance at the time the sample was received. These counters indicate the number of times the instance had become ALIVE (with `instance_state=` **DDS_ALIVE_INSTANCE_STATE** (p. 117)) at the time the sample was received.
- The `sample_rank` and `generation_rank` of the sample within the returned sequence. These ranks provide a preview of the samples that follow within the sequence returned by the `read` or `take` operations.
- The `absolute_generation_rank` of the sample within the **DDSDataReader** (p. 1087). This rank provides a preview of what is available within the **DDSDataReader** (p. 1087).
- The `source_timestamp` of the sample. This is the timestamp provided by the **DDSDataWriter** (p. 1113) at the time the sample was produced.

6.119.3 Interpretation of the SampleInfo disposed_generation_count and no_writers_generation_count

For each instance, RTI Connext internally maintains two counts, the **DDS_SampleInfo::disposed_generation_count** (p. 918) and **DDS_SampleInfo::no_writers_generation_count** (p. 918), relative to each DataReader:

- The **DDS_SampleInfo::disposed_generation_count** (p. 918) and **DDS_SampleInfo::no_writers_generation_count** (p. 918) are initialized to zero when the **DDSDataReader** (p. 1087) first detects the presence of a never-seen-before instance.
- The **DDS_SampleInfo::disposed_generation_count** (p. 918) is incremented each time the instance_state of the corresponding instance changes from **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) to **DDS_ALIVE_INSTANCE_STATE** (p. 117).

- The **DDS_SampleInfo::no_writers_generation_count** (p. 918) is incremented each time the instance_state of the corresponding instance changes from **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117) to **DDS_ALIVE_INSTANCE_STATE** (p. 117).

- These 'generation counts' are reset to zero when the instance resource is reclaimed.

The **DDS_SampleInfo::disposed_generation_count** (p. 918) and **DDS_SampleInfo::no_writers_generation_count** (p. 918) available in the **DDS_SampleInfo** (p. 912) capture a snapshot of the corresponding counters at the time the sample was received.

6.119.4 Interpretation of the SampleInfo sample_rank, generation_rank and absolute_generation_rank

The **DDS_SampleInfo::sample_rank** (p. 918) and **DDS_SampleInfo::generation_rank** (p. 918) available in the **DDS_SampleInfo** (p. 912) are computed based solely on the actual samples in the ordered collection returned by read or take.

- The **DDS_SampleInfo::sample_rank** (p. 918) indicates the number of samples of the same instance that follow the current one in the collection.

- The **DDS_SampleInfo::generation_rank** (p. 918) available in the **DDS_SampleInfo** (p. 912) indicates the difference in "generations" between the sample (S) and the Most Recent Sample of the same instance that appears in the returned Collection (MRSIC). That is, it counts the number of times the instance transitioned from not-alive to alive in the time from the reception of the S to the reception of MRSIC.

- These 'generation ranks' are reset to zero when the instance resource is reclaimed.

The **DDS_SampleInfo::generation_rank** (p. 918) is computed using the formula:

```
generation_rank = (MRSIC.disposed_generation_count
                        + MRSIC.no_writers_generation_count)
                    - (S.disposed_generation_count
                        + S.no_writers_generation_count)
```

The **DDS_SampleInfo::absolute_generation_rank** (p. 919) available in the **DDS_SampleInfo** (p. 912) indicates the difference in "generations" between

the sample (S) and the Most Recent Sample of the same instance that the middleware has received (MRS). That is, it counts the number of times the instance transitioned from not-alive to alive in the time from the reception of the S to the time when the read or take was called.

```
absolute_generation_rank = (MRS.disposed_generation_count
                                + MRS.no_writers_generation_count)
                            - (S.disposed_generation_count
                                + S.no_writers_generation_count)
```

6.119.5 Interpretation of the SampleInfo counters and ranks

These counters and ranks allow the application to distinguish samples belonging to different "generations" of the instance. Note that it is possible for an instance to transition from not-alive to alive (and back) several times before the application accesses the data by means of read or take. In this case, the returned collection may contain samples that cross generations (i.e. some samples were received before the instance became not-alive, other after the instance re-appeared again). Using the information in the **DDS_SampleInfo** (p. 912), the application can anticipate what other information regarding the same instance appears in the returned collection, as well as in the infrastructure and thus make appropriate decisions.

For example, an application desiring to only consider the most current sample for each instance would only look at samples with sample_rank == 0. Similarly, an application desiring to only consider samples that correspond to the latest generation in the collection will only look at samples with generation_rank == 0. An application desiring only samples pertaining to the latest generation available will ignore samples for which absolute_generation_rank != 0. Other application-defined criteria may also be used.

See also:

DDS_SampleStateKind (p. 111), **DDS_InstanceStateKind** (p. 116), **DDS_ViewStateKind** (p. 113), **DDS_SampleInfo::valid_data** (p. 919)

6.119.6 Member Data Documentation

6.119.6.1 DDS_SampleStateKind DDS_SampleInfo::sample_state

The sample state of the sample.

Indicates whether or not the corresponding data sample has already been read.

See also:

DDS_SampleStateKind (p. 111)

6.119.6.2 DDS_ViewStateKind DDS_SampleInfo::view_state

The view state of the instance.

Indicates whether the **DDSDataReader** (p. 1087) has already seen samples for the most-current generation of the related instance.

See also:

DDS_ViewStateKind (p. 113)

6.119.6.3 DDS_InstanceStateKind DDS_SampleInfo::instance_state

The instance state of the instance.

Indicates whether the instance is currently in existence or, if it has been disposed, the reason why it was disposed.

See also:

DDS_InstanceStateKind (p. 116)

6.119.6.4 struct DDS_Time_t DDS_SampleInfo::source_timestamp [read]

The timestamp when the sample was written by a DataWriter.

6.119.6.5 DDS_InstanceHandle_t DDS_SampleInfo::instance_handle

Identifies locally the corresponding instance.

6.119.6.6 DDS_InstanceHandle_t DDS_SampleInfo::publication_handle

Identifies locally the DataWriter that modified the instance.

The `publication_handle` is the same **DDS_InstanceHandle_t** (p. 53) that is returned by the operation **DDSDataReader::get_matched_publications** (p. 1096) and can also be used as a parameter to the operation **DDSDataReader::get_matched_publication_data** (p. 1097).

6.119.6.7 DDS_Long DDS_SampleInfo::disposed_generation_count

The disposed generation count of the instance at the time of sample reception.

Indicates the number of times the instance had become alive after it was disposed explicitly by a **DDSDataWriter** (p. 1113), at the time the sample was received.

See also:

Interpretation of the SampleInfo disposed_generation_count and no_writers_generation_count (p. 914) **Interpretation of the SampleInfo counters and ranks** (p. 916)

6.119.6.8 DDS_Long DDS_SampleInfo::no_writers_generation_count

The no writers generation count of the instance at the time of sample reception.

Indicates the number of times the instance had become alive after it was disposed because there were no writers, at the time the sample was received.

See also:

Interpretation of the SampleInfo disposed_generation_count and no_writers_generation_count (p. 914) **Interpretation of the SampleInfo counters and ranks** (p. 916)

6.119.6.9 DDS_Long DDS_SampleInfo::sample_rank

The sample rank of the sample.

Indicates the number of samples related to the same instance that follow in the collection returned by `read` or `take`.

See also:

Interpretation of the SampleInfo sample_rank, generation_rank and absolute_generation_rank (p. 915) **Interpretation of the SampleInfo counters and ranks** (p. 916)

6.119.6.10 DDS_Long DDS_SampleInfo::generation_rank

The generation rank of the sample.

Indicates the generation difference (number of times the instance was disposed and become alive again) between the time the sample was received, and the

time the most recent sample in the collection related to the same instance was received.

See also:

Interpretation of the SampleInfo sample_rank, generation_rank and absolute_generation_rank (p. 915) **Interpretation of the SampleInfo counters and ranks** (p. 916)

6.119.6.11 DDS_Long DDS_SampleInfo::absolute_generation_rank

The absolute generation rank of the sample.

Indicates the generation difference (number of times the instance was disposed and become alive again) between the time the sample was received, and the time the most recent sample (which may not be in the returned collection) related to the same instance was received.

See also:

Interpretation of the SampleInfo sample_rank, generation_rank and absolute_generation_rank (p. 915) **Interpretation of the SampleInfo counters and ranks** (p. 916)

6.119.6.12 DDS_Boolean DDS_SampleInfo::valid_data

Indicates whether the `DataSample` contains data or else it is only used to communicate a change in the `instance_state` of the instance.

Normally each `DataSample` contains both a **DDS_SampleInfo** (p. 912) and some Data. However there are situations where a `DataSample` contains only the **DDS_SampleInfo** (p. 912) and does not have any associated data. This occurs when the RTI Connext notifies the application of a change of state for an instance that was caused by some internal mechanism (such as a timeout) for which there is no associated data. An example of this situation is when the RTI Connext detects that an instance has no writers and changes the corresponding instance_state to **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117).

The application can distinguish whether a particular `DataSample` has data by examining the value of the `valid_data` flag. If this flag is set to **DDS_BOOLEAN_TRUE** (p. 298), then the `DataSample` contains valid Data. If the flag is set to **DDS_BOOLEAN_FALSE** (p. 299), the `DataSample` contains no Data.

To ensure correctness and portability, the `valid_data` flag must be examined by the application prior to accessing the Data associated with the `DataSample` and

if the flag is set to **DDS_BOOLEAN_FALSE** (p. 299), the application should not access the Data associated with the `DataSample`, that is, the application should access only the **DDS_SampleInfo** (p. 912).

6.119.6.13 struct DDS_Time_t DDS_SampleInfo::reception_timestamp [read]

<<*eXtension*>> *(p. 199)* The timestamp when the sample was committed by a DataReader.

6.119.6.14 struct DDS_SequenceNumber_t DDS_SampleInfo::publication_sequence_number [read]

<<*eXtension*>> *(p. 199)* The publication sequence number.

6.119.6.15 struct DDS_SequenceNumber_t DDS_SampleInfo::reception_sequence_number [read]

<<*eXtension*>> *(p. 199)* The reception sequence number when sample was committed by a DataReader

6.119.6.16 struct DDS_GUID_t DDS_SampleInfo::original_publication_virtual_guid [read]

<<*eXtension*>> *(p. 199)* The original publication virtual GUID.

If the **DDS_PresentationQosPolicy::access_scope** (p. 826) of the **DDSPublisher** (p. 1346) is **DDS_GROUP_PRESENTATION_QOS** (p. 352), this field contains the **DDSPublisher** (p. 1346) virtual GUID that uniquely identifies the DataWriter group.

6.119.6.17 struct DDS_SequenceNumber_t DDS_SampleInfo::original_publication_virtual_sequence_number [read]

<<*eXtension*>> *(p. 199)* The original publication virtual sequence number.

If the **DDS_PresentationQosPolicy::access_scope** (p. 826) of the **DDSPublisher** (p. 1346) is **DDS_GROUP_PRESENTATION_QOS** (p. 352), this field contains the **DDSPublisher** (p. 1346) virtual sequence number that uniquely identifies a sample within the DataWriter group.

6.119.6.18 struct DDS_GUID_t DDS_SampleInfo::related_original_publication_virtual_guid [read]

<<*eXtension*>> *(*p. *199)* The original publication virtual GUID of a related sample

6.119.6.19 struct DDS_SequenceNumber_t DDS_SampleInfo::related_original_publication_virtual_sequence_number [read]

<<*eXtension*>> *(*p. *199)* The original publication virtual sequence number of a related sample

6.120 DDS_SampleInfoSeq Struct Reference

Declares IDL `sequence` < **DDS_SampleInfo** (p. 912) > .

6.120.1 Detailed Description

Declares IDL `sequence` < **DDS_SampleInfo** (p. 912) > .

See also:

FooSeq (p. 1494)

Examples:

HelloWorld_subscriber.cxx.

6.121 DDS_SampleLostStatus Struct Reference

DDS_SAMPLE_LOST_STATUS (p. 324)

Public Attributes

- **DDS_Long total_count**

  Total cumulative count of all samples lost across all instances of data published under the ***DDSTopic*** *(p. 1419).*

- **DDS_Long total_count_change**

  The incremental number of samples lost since the last time the listener was called or the status was read.

- **DDS_SampleLostStatusKind last_reason**

  Reason why the last sample was lost.

6.121.1 Detailed Description

DDS_SAMPLE_LOST_STATUS (p. 324)

Examples:

HelloWorld_subscriber.cxx.

6.121.2 Member Data Documentation

6.121.2.1 DDS_Long DDS_SampleLostStatus::total_count

Total cumulative count of all samples lost across all instances of data published under the **DDSTopic** (p. 1419).

6.121.2.2 DDS_Long DDS_SampleLostStatus::total_count_change

The incremental number of samples lost since the last time the listener was called or the status was read.

6.121.2.3 DDS_SampleLostStatusKind DDS_SampleLostStatus::last_reason

Reason why the last sample was lost.

See also:

DDS_SampleLostStatusKind (p. 103)

6.122 DDS_SampleRejectedStatus Struct Reference

DDS_SAMPLE_REJECTED_STATUS (p. 324)

Public Attributes

- **DDS_Long total_count**

  *Total cumulative count of samples rejected by the **DDSDataReader** (p. 1087).*

- **DDS_Long total_count_change**

  The incremental number of samples rejected since the last time the listener was called or the status was read.

- **DDS_SampleRejectedStatusKind last_reason**

  Reason for rejecting the last sample rejected.

- **DDS_InstanceHandle_t last_instance_handle**

  Handle to the instance being updated by the last sample that was rejected.

6.122.1 Detailed Description

DDS_SAMPLE_REJECTED_STATUS (p. 324)

Examples:

HelloWorld_subscriber.cxx.

6.122.2 Member Data Documentation

6.122.2.1 DDS_Long DDS_SampleRejectedStatus::total_count

Total cumulative count of samples rejected by the **DDSDataReader** (p. 1087).

6.122.2.2 DDS_Long DDS_SampleRejectedStatus::total_count_change

The incremental number of samples rejected since the last time the listener was called or the status was read.

6.122.2.3 DDS_SampleRejectedStatusKind DDS_SampleRejectedStatus::last_reason

Reason for rejecting the last sample rejected.

See also:

DDS_SampleRejectedStatusKind (p. 105)

6.122.2.4 DDS_InstanceHandle_t DDS_SampleRejectedStatus::last_instance_handle

Handle to the instance being updated by the last sample that was rejected.

6.123 DDS_SequenceNumber_t Struct Reference

Type for *sequence* number representation.

Public Attributes

- **DDS_Long high**

  The most significant part of the sequence number.

- **DDS_UnsignedLong low**

  The least significant part of the sequence number.

6.123.1 Detailed Description

Type for *sequence* number representation.

Represents a 64-bit sequence number.

6.123.2 Member Data Documentation

6.123.2.1 DDS_Long DDS_SequenceNumber_t::high

The most significant part of the sequence number.

6.123.2.2 DDS_UnsignedLong DDS_SequenceNumber_t::low

The least significant part of the sequence number.

6.124 DDS_ShortSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_Short** (p. 299) >.

6.124.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_Short** (p. 299) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Short (p. 299)
FooSeq (p. 1494)

6.125 DDS_StringSeq Struct Reference

Instantiates FooSeq (p. 1494) < char* > with value type semantics.

6.125.1 Detailed Description

Instantiates FooSeq (p. 1494) < char* > with value type semantics.

StringSeq is a sequence that contains strings.

Even though the element type is a char*, i.e. a pointer, the sequence semantically behaves as a sequence of char* *value* types. When a **DDS_StringSeq** (p. 929) is copied or deleted, the contained strings are also respectively copied or deleted.

Important: Users of this type must understand its memory management contract.

- Ownership of this sequence's buffer implies ownership of the pointers stored in that buffer; a loan of the buffer implies lack of ownership of the pointers. In other words, for a type **FooSeq** (p. 1494) where **Foo** (p. 1443) is a pointer, ownership of **Foo** (p. 1443) implies ownership of *Foo. In other words, deleting a string sequence that owns its memory implies the deletion of all strings in that sequence. See **FooSeq::loan_contiguous** (p. 1504) for more information about sequence memory ownership.
- The second important rule is that non-NULL strings are *assumed to be of sufficient size* to store the necessary characters. This is a dangerous rule, but it cannot be avoided because a string doesn't store the amount of memory it has. The only other alternative is to always free and re-allocate memory. Not only would this latter policy be very expensive, but it would essentially render any loaned **DDS_StringSeq** (p. 929) immutable, since to modify any string in it would require freeing and re-allocating that string, which would violate the first principle discussed above.

It is also worth noting that the element type of a string sequence is char*, not const char*. It is therefore incorrect and dangerous, for example, to insert a string literal into a string sequence without first copying it into mutable memory.

In order to guarantee correct behavior, it is recommended that the contained elements always be manipulated using the string support API's described in **String Support** (p. 456).

See also:

String Support (p. 456)

Instantiates:

<<***generic***>> *(*p. *199)* **FooSeq** (p. 1494)

See also:

FooSeq (p. 1494)

6.126 DDS_StructMember Struct Reference

A description of a member of a struct.

Public Attributes

- char * **name**
  The name of the struct member.
- const **DDS_TypeCode** * **type**
  The type of the struct member.
- **DDS_Boolean is_pointer**
  Indicates whether the struct member is a pointer or not.
- **DDS_Short bits**
  Number of bits of a bitfield member.
- **DDS_Boolean is_key**
  Indicates if the struct member is a key member or not.

6.126.1 Detailed Description

A description of a member of a struct.

See also:

DDS_StructMemberSeq (p. 933)
DDS_TypeCodeFactory::create_struct_tc (p. 1027)

6.126.2 Member Data Documentation

6.126.2.1 char* DDS_StructMember::name

The name of the struct member.

Cannot be NULL.

6.126.2.2 const DDS_TypeCode* DDS_StructMember::type

The type of the struct member.

Cannot be NULL.

6.126.2.3 DDS_Boolean DDS_StructMember::is_pointer

Indicates whether the struct member is a pointer or not.

6.126.2.4 DDS_Short DDS_StructMember::bits

Number of bits of a bitfield member.

If the struct member is a bitfield, this field contains the number of bits of the bitfield. Otherwise, bits should contain DDS_TypeCode::NOT_BITFIELD.

6.126.2.5 DDS_Boolean DDS_StructMember::is_key

Indicates if the struct member is a key member or not.

6.127 DDS_StructMemberSeq Struct Reference

Defines a sequence of struct members.

6.127.1 Detailed Description

Defines a sequence of struct members.

See also:

DDS_StructMember (p. 931)
FooSeq (p. 1494)
DDS_TypeCodeFactory::create_struct_tc (p. 1027)

6.128 DDS_SubscriberQos Struct Reference

QoS policies supported by a **DDSSubscriber** (p. 1390) entity.

Public Attributes

- struct **DDS_PresentationQosPolicy presentation**

  Presentation policy, ***PRESENTATION*** *(p. 351).*

- struct **DDS_PartitionQosPolicy partition**

  Partition policy, ***PARTITION*** *(p. 361).*

- struct **DDS_GroupDataQosPolicy group_data**

  Group data policy, ***GROUP_DATA*** *(p. 347).*

- struct **DDS_EntityFactoryQosPolicy entity_factory**

  Entity factory policy, ***ENTITY_FACTORY*** *(p. 377).*

- struct **DDS_ExclusiveAreaQosPolicy exclusive_area**

  <<**eXtension**>> (p. 199) *Exclusive area for the subscriber and all entities that are created by the subscriber.*

6.128.1 Detailed Description

QoS policies supported by a **DDSSubscriber** (p. 1390) entity.

You must set certain members in a consistent manner:

length of DDS_SubscriberQos::group_data.value <= DDS_DomainParticipantQos::resource_limits.subscriber_group_data_max_length

length of DDS_SubscriberQos::partition.name <= DDS_DomainParticipantQos::resource_limits.max_partitions

combined number of characters (including terminating 0) in DDS_SubscriberQos::partition.name <= DDS_DomainParticipantQos::resource_limits.max_partition_cumulative_characters

If any of the above are not true, **DDSSubscriber::set_qos** (p. 1410) and **DDSSubscriber::set_qos_with_profile** (p. 1410) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.128.2 Member Data Documentation

6.128.2.1 struct DDS_PresentationQosPolicy DDS_SubscriberQos::presentation [read]

Presentation policy, **PRESENTATION** (p. 351).

6.128.2.2 struct DDS_PartitionQosPolicy DDS_SubscriberQos::partition [read]

Partition policy, **PARTITION** (p. 361).

6.128.2.3 struct DDS_GroupDataQosPolicy DDS_SubscriberQos::group_data [read]

Group data policy, **GROUP_DATA** (p. 347).

6.128.2.4 struct DDS_EntityFactoryQosPolicy DDS_SubscriberQos::entity_factory [read]

Entity factory policy, **ENTITY_FACTORY** (p. 377).

6.128.2.5 struct DDS_ExclusiveAreaQosPolicy DDS_SubscriberQos::exclusive_area [read]

<<***eXtension***>> *(p. 199)* Exclusive area for the subscriber and all entities that are created by the subscriber.

6.129 DDS_SubscriptionBuiltinTopicData Struct Reference

Entry created when a **DDSDataReader** (p. 1087) is discovered in association with its Subscriber.

Public Attributes

- **DDS_BuiltinTopicKey_t key**
  DCPS key to distinguish entries.
- **DDS_BuiltinTopicKey_t participant_key**
  DCPS key of the participant to which the DataReader belongs.
- char * **topic_name**
  *Name of the related **DDSTopic** (p. 1419).*
- char * **type_name**
  *Name of the type attached to the **DDSTopic** (p. 1419).*
- struct **DDS_DurabilityQosPolicy durability**
  Policy of the corresponding DataReader.
- struct **DDS_DeadlineQosPolicy deadline**
  Policy of the corresponding DataReader.
- struct **DDS_LatencyBudgetQosPolicy latency_budget**
  Policy of the corresponding DataReader.
- struct **DDS_LivelinessQosPolicy liveliness**
  Policy of the corresponding DataReader.
- struct **DDS_ReliabilityQosPolicy reliability**
  Policy of the corresponding DataReader.
- struct **DDS_OwnershipQosPolicy ownership**
  Policy of the corresponding DataReader.
- struct **DDS_DestinationOrderQosPolicy destination_order**
  Policy of the corresponding DataReader.
- struct **DDS_UserDataQosPolicy user_data**

Policy of the corresponding DataReader.

^ struct **DDS_TimeBasedFilterQosPolicy time_based_filter**

Policy of the corresponding DataReader.

^ struct **DDS_PresentationQosPolicy presentation**

Policy of the Subscriber to which the DataReader belongs.

^ struct **DDS_PartitionQosPolicy partition**

Policy of the Subscriber to which the DataReader belongs.

^ struct **DDS_TopicDataQosPolicy topic_data**

Policy of the related Topic.

^ struct **DDS_GroupDataQosPolicy group_data**

Policy of the Subscriber to which the DataReader belongs.

^ struct **DDS_TypeConsistencyEnforcementQosPolicy type_consistency**

Policy of the corresponding DataReader.

^ struct **DDS_TypeCode * type_code**

<<**eXtension**>> (p. 199) *Type code information of the corresponding Topic*

^ **DDS_BuiltinTopicKey_t subscriber_key**

<<**eXtension**>> (p. 199) *DCPS key of the subscriber to which the DataReader belongs.*

^ struct **DDS_PropertyQosPolicy property**

<<**eXtension**>> (p. 199) *Properties of the corresponding DataReader.*

^ struct **DDS_LocatorSeq unicast_locators**

<<**eXtension**>> (p. 199) *Custom unicast locators that the endpoint can specify. The default locators will be used if this is not specified.*

^ struct **DDS_LocatorSeq multicast_locators**

<<**eXtension**>> (p. 199) *Custom multicast locators that the endpoint can specify. The default locators will be used if this is not specified.*

^ struct **DDS_ContentFilterProperty_t content_filter_property**

<<**eXtension**>> (p. 199) *This field provides all the required information to enable content filtering on the Writer side.*

- struct **DDS_GUID_t virtual_guid**

  <<**eXtension**>> (p. 199) *Virtual GUID associated to the DataReader.*

- **DDS_ProtocolVersion_t rtps_protocol_version**

  <<**eXtension**>> (p. 199) *Version number of the RTPS wire protocol used.*

- struct **DDS_VendorId_t rtps_vendor_id**

  <<**eXtension**>> (p. 199) *ID of vendor implementing the RTPS wire protocol.*

- struct **DDS_ProductVersion_t product_version**

  <<**eXtension**>> (p. 199) *This is a vendor specific parameter. It gives the current version for rti-dds.*

- **DDS_Boolean disable_positive_acks**

  <<**eXtension**>> (p. 199) *This is a vendor specific parameter. Determines whether the corresponding DataReader sends positive acknowledgements for reliability.*

- struct **DDS_EntityNameQosPolicy subscription_name**

  <<**eXtension**>> (p. 199) *The subscription name and role name.*

6.129.1 Detailed Description

Entry created when a **DDSDataReader** (p. 1087) is discovered in association with its Subscriber.

Data associated with the built-in topic **DDS_SUBSCRIPTION_TOPIC_NAME** (p. 291). It contains QoS policies and additional information that apply to the remote **DDSDataReader** (p. 1087) the related **DDSSubscriber** (p. 1390).

See also:

DDS_SUBSCRIPTION_TOPIC_NAME (p. 291)
DDSSubscriptionBuiltinTopicDataDataReader (p. 1417)

6.129.2 Member Data Documentation

6.129.2.1 DDS_BuiltinTopicKey_t DDS_SubscriptionBuiltinTopicData::key

DCPS key to distinguish entries.

6.129.2.2 DDS_BuiltinTopicKey_t DDS_SubscriptionBuiltinTopicData::participant_key

DCPS key of the participant to which the DataReader belongs.

6.129.2.3 char* DDS_SubscriptionBuiltinTopicData::topic_name

Name of the related **DDSTopic** (p. 1419).

The length of this string is limited to 255 characters.

The memory for this field is managed as described in **Conventions** (p. 457).

See also:

Conventions (p. 457)

6.129.2.4 char* DDS_SubscriptionBuiltinTopicData::type_name

Name of the type attached to the **DDSTopic** (p. 1419).

The length of this string is limited to 255 characters.

The memory for this field is managed as described in **Conventions** (p. 457).

See also:

Conventions (p. 457)

6.129.2.5 struct DDS_DurabilityQosPolicy DDS_SubscriptionBuiltinTopicData::durability [read]

Policy of the corresponding DataReader.

6.129.2.6 struct DDS_DeadlineQosPolicy DDS_SubscriptionBuiltinTopicData::deadline [read]

Policy of the corresponding DataReader.

6.129.2.7 struct DDS_LatencyBudgetQosPolicy DDS_SubscriptionBuiltinTopicData::latency_budget [read]

Policy of the corresponding DataReader.

6.129.2.8 struct DDS_LivelinessQosPolicy DDS_SubscriptionBuiltinTopicData::liveliness [read]

Policy of the corresponding DataReader.

6.129.2.9 struct DDS_ReliabilityQosPolicy DDS_SubscriptionBuiltinTopicData::reliability [read]

Policy of the corresponding DataReader.

6.129.2.10 struct DDS_OwnershipQosPolicy DDS_SubscriptionBuiltinTopicData::ownership [read]

Policy of the corresponding DataReader.

6.129.2.11 struct DDS_DestinationOrderQosPolicy DDS_SubscriptionBuiltinTopicData::destination_order [read]

Policy of the corresponding DataReader.

6.129.2.12 struct DDS_UserDataQosPolicy DDS_SubscriptionBuiltinTopicData::user_data [read]

Policy of the corresponding DataReader.

6.129.2.13 struct DDS_TimeBasedFilterQosPolicy DDS_SubscriptionBuiltinTopicData::time_based_filter [read]

Policy of the corresponding DataReader.

6.129.2.14 struct DDS_PresentationQosPolicy DDS_SubscriptionBuiltinTopicData::presentation [read]

Policy of the Subscriber to which the DataReader belongs.

6.129.2.15 struct DDS_PartitionQosPolicy DDS_SubscriptionBuiltinTopicData::partition [read]

Policy of the Subscriber to which the DataReader belongs.

6.129.2.16 struct DDS_TopicDataQosPolicy DDS_SubscriptionBuiltinTopicData::topic_data [read]

Policy of the related Topic.

6.129.2.17 struct DDS_GroupDataQosPolicy DDS_SubscriptionBuiltinTopicData::group_data [read]

Policy of the Subscriber to which the DataReader belongs.

6.129.2.18 struct DDS_TypeConsistencyEnforcementQosPolicy DDS_SubscriptionBuiltinTopicData::type_consistency [read]

Policy of the corresponding DataReader.

6.129.2.19 struct DDS_TypeCode* DDS_SubscriptionBuiltinTopicData::type_code [read]

<<***eXtension***>> *(p. 199)* Type code information of the corresponding Topic

6.129.2.20 DDS_BuiltinTopicKey_t DDS_SubscriptionBuiltinTopicData::subscriber_key

<<***eXtension***>> *(p. 199)* DCPS key of the subscriber to which the DataReader belongs.

6.129.2.21 struct DDS_PropertyQosPolicy DDS_SubscriptionBuiltinTopicData::property [read]

<<*eXtension*>> *(p. 199)* Properties of the corresponding DataReader.

6.129.2.22 struct DDS_LocatorSeq DDS_SubscriptionBuiltinTopicData::unicast_locators [read]

<<*eXtension*>> *(p. 199)* Custom unicast locators that the endpoint can specify. The default locators will be used if this is not specified.

6.129.2.23 struct DDS_LocatorSeq DDS_SubscriptionBuiltinTopicData::multicast_locators [read]

<<*eXtension*>> *(p. 199)* Custom multicast locators that the endpoint can specify. The default locators will be used if this is not specified.

6.129.2.24 struct DDS_ContentFilterProperty_t DDS_SubscriptionBuiltinTopicData::content_filter_property [read]

<<*eXtension*>> *(p. 199)* This field provides all the required information to enable content filtering on the Writer side.

6.129.2.25 struct DDS_GUID_t DDS_SubscriptionBuiltinTopicData::virtual_guid [read]

<<*eXtension*>> *(p. 199)* Virtual GUID associated to the DataReader.

See also:

DDS_GUID_t (p. 757)

6.129.2.26 DDS_ProtocolVersion_t DDS_SubscriptionBuiltinTopicData::rtps_protocol_version

<<*eXtension*>> *(p. 199)* Version number of the RTPS wire protocol used.

6.129.2.27 struct DDS_VendorId_t DDS_SubscriptionBuiltinTopicData::rtps_vendor_id [read]

<<*eXtension*>> *(*p. *199)* ID of vendor implementing the RTPS wire protocol.

6.129.2.28 struct DDS_ProductVersion_t DDS_SubscriptionBuiltinTopicData::product_version [read]

<<*eXtension*>> *(*p. *199)* This is a vendor specific parameter. It gives the current version for rti-dds.

6.129.2.29 DDS_Boolean DDS_SubscriptionBuiltinTopicData::disable_positive_acks

<<*eXtension*>> *(*p. *199)* This is a vendor specific parameter. Determines whether the corresponding DataReader sends positive acknowledgements for reliability.

6.129.2.30 struct DDS_EntityNameQosPolicy DDS_SubscriptionBuiltinTopicData::subscription_name [read]

<<*eXtension*>> *(*p. *199)* The subscription name and role name.

This member contains the name and the role name of the discovered subscription.

6.130 DDS_SubscriptionBuiltinTopicDataSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_SubscriptionBuiltinTopicData** (p. 936) > .

6.130.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_SubscriptionBuiltinTopicData** (p. 936) > .

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_SubscriptionBuiltinTopicData (p. 936)

6.131 DDS_SubscriptionMatchedStatus Struct Reference

DDS_SUBSCRIPTION_MATCHED_STATUS (p. 326)

Public Attributes

- **DDS_Long total_count**

  *The total cumulative number of times the concerned **DDSDataReader** (p. 1087) discovered a "match" with a **DDSDataWriter** (p. 1113).*

- **DDS_Long total_count_change**

  The change in total_count since the last time the listener was called or the status was read.

- **DDS_Long current_count**

  *The current number of writers with which the **DDSDataReader** (p. 1087) is matched.*

- **DDS_Long current_count_peak**

  <<**eXtension**>> (p. 199) *The highest value that current_count has reached until now.*

- **DDS_Long current_count_change**

  The change in current_count since the last time the listener was called or the status was read.

- **DDS_InstanceHandle_t last_publication_handle**

  *A handle to the last **DDSDataWriter** (p. 1113) that caused the status to change.*

6.131.1 Detailed Description

DDS_SUBSCRIPTION_MATCHED_STATUS (p. 326)

A "match" happens when the **DDSDataReader** (p. 1087) finds a **DDSDataWriter** (p. 1113) for the same **DDSTopic** (p. 1419) with an offered QoS that is compatible with that requested by the **DDSDataReader** (p. 1087).

This status is also changed (and the listener, if any, called) when a match is ended. A local **DDSDataReader** (p. 1087) will become "unmatched" from a remote **DDSDataWriter** (p. 1113) when that **DDSDataWriter** (p. 1113) goes away for any reason.

Examples:

HelloWorld_subscriber.cxx.

6.131.2 Member Data Documentation

6.131.2.1 DDS_Long DDS_SubscriptionMatchedStatus::total_count

The total cumulative number of times the concerned **DDSDataReader** (p. 1087) discovered a "match" with a **DDSDataWriter** (p. 1113).

This number increases whenever a new match is discovered. It does not change when an existing match goes away.

6.131.2.2 DDS_Long DDS_SubscriptionMatchedStatus::total_count_change

The change in total_count since the last time the listener was called or the status was read.

6.131.2.3 DDS_Long DDS_SubscriptionMatchedStatus::current_count

The current number of writers with which the **DDSDataReader** (p. 1087) is matched.

This number increases when a new match is discovered and decreases when an existing match goes away.

6.131.2.4 DDS_Long DDS_SubscriptionMatchedStatus::current_count_peak

<<***eXtension***>> *(p. 199)* The highest value that current_count has reached until now.

6.131.2.5 DDS_Long DDS_SubscriptionMatchedStatus::current_count_change

The change in current_count since the last time the listener was called or the status was read.

6.131.2.6 DDS_InstanceHandle_t DDS_SubscriptionMatchedStatus::last_publication_handle

A handle to the last **DDSDataWriter** (p. 1113) that caused the status to change.

6.132 DDS_SystemResourceLimitsQosPolicy Struct Reference

Configures **DDSDomainParticipant** (p. 1139)-independent resources used by RTI Connext. Mainly used to change the maximum number of **DDSDomainParticipant** (p. 1139) entities that can be created within a single process (address space).

Public Attributes

- **DDS_Long max_objects_per_thread**

  *The maximum number of objects that can be stored per thread for a **DDSDomainParticipantFactory** (p. 1216).*

6.132.1 Detailed Description

Configures **DDSDomainParticipant** (p. 1139)-independent resources used by RTI Connext. Mainly used to change the maximum number of **DDSDomainParticipant** (p. 1139) entities that can be created within a single process (address space).

This QoS policy is an extension to the DDS standard.

Entity:

DDSDomainParticipantFactory (p. 1216)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.132.2 Usage

Within a single process (or address space for some supported real-time operating systems), applications may create and use multiple **DDSDomainParticipant** (p. 1139) entities. This QoS policy sets a parameter that places an effective upper bound on the maximum number of **DDSDomainParticipant** (p. 1139) entities that can be created in a single process/address space.

6.132.3 Member Data Documentation

6.132.3.1 DDS_Long DDS_SystemResourceLimitsQosPolicy::max_objects_per_thread

The maximum number of objects that can be stored per thread for a **DDSDomainParticipantFactory** (p. 1216).

Before increasing this value to allow you to create more participants, carefully consider the application design that requires you to create so many participants. Remember: a **DDSDomainParticipant** (p. 1139) is a heavy-weight object. It spawns several threads and maintains its own discovery database (see **DISCOVERY** (p. 387)). Creating more participants than RTI Connext strictly requires – one per domain per process/address space – can adversely affect the performance and resource utilization of your application.

[**default**] 1024; this value allows you to create about 10 or 11 **DDSDomainParticipant** (p. 1139) entities.

[**range**] [1, 1 billion]

6.133 DDS_ThreadSettings_t Struct Reference

The properties of a thread of execution.

Public Attributes

- **DDS_ThreadSettingsKindMask mask**

  Describes the type of thread.

- **DDS_Long priority**

  Thread priority.

- **DDS_Long stack_size**

  The thread stack-size.

- struct **DDS_LongSeq cpu_list**

  The list of processors on which the thread(s) may run.

- **DDS_ThreadSettingsCpuRotationKind cpu_rotation**

  Determines how processor affinity is applied to multiple threads.

6.133.1 Detailed Description

The properties of a thread of execution.

QoS:

DDS_EventQosPolicy (p. 739) **DDS_DatabaseQosPolicy** (p. 495) **DDS_ReceiverPoolQosPolicy** (p. 862) **DDS_AsynchronousPublisherQosPolicy** (p. 466)

6.133.2 Member Data Documentation

6.133.2.1 DDS_ThreadSettingsKindMask DDS_ThreadSettings_t::mask

Describes the type of thread.

[default] 0, use default options of the OS

6.133.2.2 DDS_Long DDS_ThreadSettings_t::priority

Thread priority.

[**range**] Platform-dependent

6.133.2.3 DDS_Long DDS_ThreadSettings_t::stack_size

The thread stack-size.

[**range**] Platform-dependent.

6.133.2.4 struct DDS_LongSeq DDS_ThreadSettings_t::cpu_list [read]

The list of processors on which the thread(s) may run.

A sequence of integers that represent the set of processors on which the thread(s) controlled by this QoS may run. An empty sequence (the default) means the middleware wlll make no CPU affinity adjustments.

Note: This feature is currently only supported on a subset of architectures (see the `Platform Notes`). The API may change as more architectures are added in future releases.

This value is only relevant to the **DDS_ReceiverPoolQosPolicy** (p. 862). It is ignored within other QoS policies that include **DDS_ThreadSettings_t** (p. 950).

See also:

Controlling CPU Core Affinity for RTI Threads (p. 330)

[**default**] Empty sequence

6.133.2.5 DDS_ThreadSettingsCpuRotationKind DDS_ThreadSettings_t::cpu_rotation

Determines how processor affinity is applied to multiple threads.

This value is only relevant to the **DDS_ReceiverPoolQosPolicy** (p. 862). It is ignored within other QoS policies that include **DDS_ThreadSettings_t** (p. 950).

See also:

Controlling CPU Core Affinity for RTI Threads (p. 330)

Note: This feature is currently only supported on a subset of architectures (see the `Platform Notes`). The API may change as more architectures are added in future releases.;

6.134 DDS_Time_t Struct Reference

Type for *time* representation.

Public Attributes

- **DDS_Long sec**
  seconds
- **DDS_UnsignedLong nanosec**
  nanoseconds

6.134.1 Detailed Description

Type for *time* representation.

A **DDS_Time_t** (p. 953) represents a moment in time.

6.134.2 Member Data Documentation

6.134.2.1 DDS_Long DDS_Time_t::sec

seconds

6.134.2.2 DDS_UnsignedLong DDS_Time_t::nanosec

nanoseconds

6.135 DDS_TimeBasedFilterQosPolicy Struct Reference

Filter that allows a **DDSDataReader** (p. 1087) to specify that it is interested only in (potentially) a subset of the values of the data.

Public Attributes

- struct **DDS_Duration_t minimum_separation**

  The minimum separation duration between subequent samples.

6.135.1 Detailed Description

Filter that allows a **DDSDataReader** (p. 1087) to specify that it is interested only in (potentially) a subset of the values of the data.

The filter states that the **DDSDataReader** (p. 1087) does not want to receive more than one value each `minimum_separation`, regardless of how fast the changes occur.

Entity:

DDSDataReader (p. 1087)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **YES** (p. 340)

6.135.2 Usage

You can use this QoS policy to reduce the amount of data received by a **DDSDataReader** (p. 1087). **DDSDataWriter** (p. 1113) entities may send data faster than needed by a **DDSDataReader** (p. 1087). For example, a **DDSDataReader** (p. 1087) of sensor data that is displayed to a human operator in a GUI application does not need to receive data updates faster than a user can reasonably perceive changes in data values. This is often measured in tenths (0.1) of a second up to several seconds. However, a **DDSDataWriter** (p. 1113) of sensor information may have other **DDSDataReader** (p. 1087) entities that are processing the sensor information to control parts of the system and thus need new data updates in measures of hundredths (0.01) or thousandths (0.001) of a second.

With this QoS policy, different **DDSDataReader** (p. 1087) entities can set their own time-based filters, so that data published faster than the period set by a each **DDSDataReader** (p. 1087) will not be delivered to that **DDSDataReader** (p. 1087).

The **TIME_BASED_FILTER** (p. 360) also applies to each instance separately; that is, the constraint is that the **DDSDataReader** (p. 1087) does not want to see more than one sample of each instance per `minimum_separation` period.

This QoS policy allows you to optimize resource usage (CPU and possibly network bandwidth) by only delivering the required amount of data to each **DDSDataReader** (p. 1087), accommodating the fact that, for rapidly-changing data, different subscribers may have different requirements and constraints as to how frequently they need or can handle being notified of the most current values. As such, it can also be used to protect applications that are running on a heterogeneous network where some nodes are capable of generating data much faster than others can consume it.

For best effort data delivery, if the data type is unkeyed and the **DDSDataWriter** (p. 1113) has an infinite **DDS_LivelinessQosPolicy::lease_duration** (p. 782), RTI Connext will only send as many packets to a **DDSDataReader** (p. 1087) as required by the TIME_BASED_FILTER, no matter how fast **FooDataWriter::write** (p. 1484) is called.

For multicast data delivery to multiple DataReaders, the one with the lowest `minimum_separation` determines the DataWriter's send rate. For example, if a **DDSDataWriter** (p. 1113) sends over multicast to two DataReaders, one with `minimum_separation` of 2 seconds and one with `minimum_separation` of 1 second, the DataWriter will send every 1 second.

In configurations where RTI Connext must send all the data published by the **DDSDataWriter** (p. 1113) (for example, when the **DDSDataWriter** (p. 1113) is reliable, when the data type is keyed, or when the **DDSDataWriter** (p. 1113) has a finite **DDS_LivelinessQosPolicy::lease_duration** (p. 782)), only the data that passes the TIME_BASED_FILTER will be stored in the receive queue of the **DDSDataReader** (p. 1087). Extra data will be accepted but dropped. Note that filtering is only applied on alive samples (that is, samples that have not been disposed/unregistered).

6.135.3 Consistency

It is inconsistent for a **DDSDataReader** (p. 1087) to have a `minimum_separation` longer than its **DEADLINE** (p. 353) `period`.

However, it is important to be aware of certain edge cases that can occur when your publication rate, minimum separation, and deadline period align and that can cause missed deadlines that you may not expect. For example, suppose

that you nominally publish samples every second but that this rate can vary somewhat over time. You declare a minimum separation of 1 second to filter out rapid updates and set a deadline of two seconds so that you will be aware if the rate falls too low. Even if your update rate never wavers, you can still miss deadlines! Here's why:

Suppose you publish the first sample at time t=0 seconds. You then publish your next sample at t=1 seconds. Depending on how your operating system schedules the time-based filter execution relative to the publication, this second sample may be filtered. You then publish your third sample at t=2 seconds, and depending on how your OS schedules this publication in relation to the deadline check, you could miss the deadline.

This scenario demonstrates a couple of rules of thumb:

- Beware of setting your `minimum_separation` to a value very close to your publication rate: you may filter more data than you intend to.
- Beware of setting your `minimum_separation` to a value that is too close to your deadline period relative to your publication rate. You may miss deadlines.

See **DDS_DeadlineQosPolicy** (p. 567) for more information about the interactions between deadlines and time-based filters.

The setting of a **TIME_BASED_FILTER** (p. 360) – that is, the selection of a `minimum_separation` with a value greater than zero – is consistent with all settings of the **HISTORY** (p. 367) and **RELIABILITY** (p. 362) QoS. The **TIME_BASED_FILTER** (p. 360) specifies the samples that are of interest to the **DDSDataReader** (p. 1087). The **HISTORY** (p. 367) and **RELIABILITY** (p. 362) QoS affect the behavior of the middleware with respect to the samples that have been determined to be of interest to the **DDSDataReader** (p. 1087); that is, they apply *after* the **TIME_BASED_FILTER** (p. 360) has been applied.

In the case where the reliability QoS kind is **DDS_RELIABLE_RELIABILITY_QOS** (p. 363), in steady-state – defined as the situation where the **DDSDataWriter** (p. 1113) does not write new samples for a period "long" compared to the `minimum_separation` – the system should guarantee delivery of the last sample to the **DDSDataReader** (p. 1087).

See also:

DeadlineQosPolicy
HistoryQosPolicy
ReliabilityQosPolicy

6.135.4 Member Data Documentation

6.135.4.1 struct DDS_Duration_t DDS_TimeBasedFilterQosPolicy::minimum_separation [read]

The minimum separation duration between subequent samples.

[**default**] 0 (meaning the **DDSDataReader** (p. 1087) is potentially interested in all values)

[**range**] [0,1 year], < **DDS_DeadlineQosPolicy::period** (p. 569)

6.136 DDS_TopicBuiltinTopicData Struct Reference

Entry created when a Topic object discovered.

Public Attributes

- **DDS_BuiltinTopicKey_t key**
  DCPS key to distinguish entries.
- char * **name**
  *Name of the **DDSTopic** (p. 1419).*
- char * **type_name**
  *Name of the type attached to the **DDSTopic** (p. 1419).*
- struct **DDS_DurabilityQosPolicy durability**
  durability policy of the corresponding Topic
- struct **DDS_DurabilityServiceQosPolicy durability_service**
  durability service policy of the corresponding Topic
- struct **DDS_DeadlineQosPolicy deadline**
  Policy of the corresponding Topic.
- struct **DDS_LatencyBudgetQosPolicy latency_budget**
  Policy of the corresponding Topic.
- struct **DDS_LivelinessQosPolicy liveliness**
  Policy of the corresponding Topic.
- struct **DDS_ReliabilityQosPolicy reliability**
  Policy of the corresponding Topic.
- struct **DDS_TransportPriorityQosPolicy transport_priority**
  Policy of the corresponding Topic.
- struct **DDS_LifespanQosPolicy lifespan**
  Policy of the corresponding Topic.
- struct **DDS_DestinationOrderQosPolicy destination_order**
  Policy of the corresponding Topic.

- struct **DDS_HistoryQosPolicy history**

  Policy of the corresponding Topic.

- struct **DDS_ResourceLimitsQosPolicy resource_limits**

  Policy of the corresponding Topic.

- struct **DDS_OwnershipQosPolicy ownership**

  Policy of the corresponding Topic.

- struct **DDS_TopicDataQosPolicy topic_data**

  Policy of the corresponding Topic.

6.136.1 Detailed Description

Entry created when a Topic object discovered.

Data associated with the built-in topic **DDS_TOPIC_TOPIC_NAME** (p. 287). It contains QoS policies and additional information that apply to the remote **DDSTopic** (p. 1419).

Note: The **DDS_TopicBuiltinTopicData** (p. 958) built-in topic is meant to convey information about discovered Topics. This Topic's samples are not propagated in a separate packet on the wire. Instead, the data is sent as part of the information carried by other built-in topics (**DDS_PublicationBuiltinTopicData** (p. 839) and **DDS_SubscriptionBuiltinTopicData** (p. 936)). Therefore TopicBuiltinTopicData DataReaders will not receive any data.

See also:

DDS_TOPIC_TOPIC_NAME (p. 287)
DDSTopicBuiltinTopicDataDataReader (p. 1425)

6.136.2 Member Data Documentation

6.136.2.1 DDS_BuiltinTopicKey_t DDS_TopicBuiltinTopicData::key

DCPS key to distinguish entries.

6.136.2.2 char* DDS_TopicBuiltinTopicData::name

Name of the **DDSTopic** (p. 1419).

The length of this string is limited to 255 characters.

The memory for this field is managed as described in **Conventions** (p. 457).

See also:

Conventions (p. 457)

6.136.2.3 char* DDS_TopicBuiltinTopicData::type_name

Name of the type attached to the **DDSTopic** (p. 1419).

The length of this string is limited to 255 characters.

The memory for this field is managed as described in **Conventions** (p. 457).

See also:

Conventions (p. 457)

6.136.2.4 struct DDS_DurabilityQosPolicy DDS_TopicBuiltinTopicData::durability [read]

durability policy of the corresponding Topic

6.136.2.5 struct DDS_DurabilityServiceQosPolicy DDS_TopicBuiltinTopicData::durability_service [read]

durability service policy of the corresponding Topic

6.136.2.6 struct DDS_DeadlineQosPolicy DDS_TopicBuiltinTopicData::deadline [read]

Policy of the corresponding Topic.

6.136.2.7 struct DDS_LatencyBudgetQosPolicy DDS_TopicBuiltinTopicData::latency_budget [read]

Policy of the corresponding Topic.

6.136.2.8 struct DDS_LivelinessQosPolicy DDS_TopicBuiltinTopicData::liveliness [read]

Policy of the corresponding Topic.

6.136.2.9 struct DDS_ReliabilityQosPolicy DDS_TopicBuiltinTopicData::reliability [read]

Policy of the corresponding Topic.

6.136.2.10 struct DDS_TransportPriorityQosPolicy DDS_TopicBuiltinTopicData::transport_priority [read]

Policy of the corresponding Topic.

6.136.2.11 struct DDS_LifespanQosPolicy DDS_TopicBuiltinTopicData::lifespan [read]

Policy of the corresponding Topic.

6.136.2.12 struct DDS_DestinationOrderQosPolicy DDS_TopicBuiltinTopicData::destination_order [read]

Policy of the corresponding Topic.

6.136.2.13 struct DDS_HistoryQosPolicy DDS_TopicBuiltinTopicData::history [read]

Policy of the corresponding Topic.

6.136.2.14 struct DDS_ResourceLimitsQosPolicy DDS_TopicBuiltinTopicData::resource_limits [read]

Policy of the corresponding Topic.

6.136.2.15 struct DDS_OwnershipQosPolicy DDS_TopicBuiltinTopicData::ownership [read]

Policy of the corresponding Topic.

6.136.2.16 struct DDS_TopicDataQosPolicy DDS_TopicBuiltinTopicData::topic_data [read]

Policy of the corresponding Topic.

6.137 DDS_TopicBuiltinTopicDataSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_TopicBuiltinTopicData** (p. 958) > .

6.137.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_TopicBuiltinTopicData** (p. 958) > .

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_TopicBuiltinTopicData (p. 958)

6.138 DDS_TopicDataQosPolicy Struct Reference

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Public Attributes

- struct **DDS_OctetSeq value**
  a sequence of octets

6.138.1 Detailed Description

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Entity:

DDSTopic (p. 1419)

Properties:

RxO (p. 340) = NO
Changeable (p. 340) = **YES** (p. 340)

See also:

DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.138.2 Usage

The purpose of this QoS is to allow the application to attach additional information to the created **DDSTopic** (p. 1419) objects, so that when a remote application discovers their existence, it can access that information and use it for its own purposes. This extra data is not used by RTI Connext.

One possible use of this QoS is to attach security credentials or some other information that can be used by the remote application to authenticate the source.

In combination with **DDSDataReaderListener** (p. 1108), **DDSDataWriterListener** (p. 1133), or operations such as **DDSDomainParticipant::ignore_topic** (p. 1188), this QoS policy can assist an application in defining and enforcing its own security policies.

The use of this QoS is not limited to security; it offers a simple, yet flexible extensibility mechanism.

Important: RTI Connext stores the data placed in this policy in pre-allocated pools. It is therefore necessary to configure RTI Connext with the maximum size of the data that will be stored in policies of this type. This size is configured with **DDS_DomainParticipantResourceLimitsQosPolicy::topic_data_max_length** (p. 606).

6.138.3 Member Data Documentation

6.138.3.1 struct DDS_OctetSeq DDS_TopicDataQosPolicy::value [read]

a sequence of octets

[default] empty (zero-length)

[range] Octet sequence of length [0,max_length]

6.139 DDS_TopicQos Struct Reference

QoS policies supported by a **DDSTopic** (p. 1419) entity.

Public Attributes

- struct **DDS_TopicDataQosPolicy topic_data**
  Topic data policy, ***TOPIC_DATA*** *(p. 346).*
- struct **DDS_DurabilityQosPolicy durability**
  Durability policy, ***DURABILITY*** *(p. 348).*
- struct **DDS_DurabilityServiceQosPolicy durability_service**
  DurabilityService policy, ***DURABILITY_SERVICE*** *(p. 370).*
- struct **DDS_DeadlineQosPolicy deadline**
  Deadline policy, ***DEADLINE*** *(p. 353).*
- struct **DDS_LatencyBudgetQosPolicy latency_budget**
  Latency budget policy, ***LATENCY_BUDGET*** *(p. 354).*
- struct **DDS_LivelinessQosPolicy liveliness**
  Liveliness policy, ***LIVELINESS*** *(p. 358).*
- struct **DDS_ReliabilityQosPolicy reliability**
  Reliability policy, ***RELIABILITY*** *(p. 362).*
- struct **DDS_DestinationOrderQosPolicy destination_order**
  Destination order policy, ***DESTINATION_ORDER*** *(p. 365).*
- struct **DDS_HistoryQosPolicy history**
  History policy, ***HISTORY*** *(p. 367).*
- struct **DDS_ResourceLimitsQosPolicy resource_limits**
  Resource limits policy, ***RESOURCE_LIMITS*** *(p. 371).*
- struct **DDS_TransportPriorityQosPolicy transport_priority**
  Transport priority policy, ***TRANSPORT_PRIORITY*** *(p. 373).*
- struct **DDS_LifespanQosPolicy lifespan**
  Lifespan policy, ***LIFESPAN*** *(p. 374).*

^ struct **DDS_OwnershipQosPolicy ownership**

*Ownership policy, **OWNERSHIP** (p. 355).*

6.139.1 Detailed Description

QoS policies supported by a **DDSTopic** (p. 1419) entity.

You must set certain members in a consistent manner:

length of **DDS_TopicQos::topic_data** (p. 966) .value <= **DDS_DomainParticipantQos::resource_limits** (p. 591) .topic_data_max_length

If any of the above are not true, **DDSTopic::set_qos** (p. 1421), **DDSTopic::set_qos_with_profile** (p. 1422) and **DDSDomainParticipant::set_default_topic_qos** (p. 1162) will fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) and **DDSDomainParticipant::create_topic** (p. 1175) will return NULL.

Entity:

DDSTopic (p. 1419)

See also:

QoS Policies (p. 331) allowed ranges within each Qos.

6.139.2 Member Data Documentation

6.139.2.1 struct DDS_TopicDataQosPolicy DDS_TopicQos::topic_data [read]

Topic data policy, **TOPIC_DATA** (p. 346).

6.139.2.2 struct DDS_DurabilityQosPolicy DDS_TopicQos::durability [read]

Durability policy, **DURABILITY** (p. 348).

6.139.2.3 struct DDS_DurabilityServiceQosPolicy DDS_TopicQos::durability_service [read]

DurabilityService policy, **DURABILITY_SERVICE** (p. 370).

6.139.2.4 struct DDS_DeadlineQosPolicy DDS_TopicQos::deadline [read]

Deadline policy, **DEADLINE** (p. 353).

6.139.2.5 struct DDS_LatencyBudgetQosPolicy DDS_TopicQos::latency_budget [read]

Latency budget policy, **LATENCY_BUDGET** (p. 354).

6.139.2.6 struct DDS_LivelinessQosPolicy DDS_TopicQos::liveliness [read]

Liveliness policy, **LIVELINESS** (p. 358).

6.139.2.7 struct DDS_ReliabilityQosPolicy DDS_TopicQos::reliability [read]

Reliability policy, **RELIABILITY** (p. 362).

6.139.2.8 struct DDS_DestinationOrderQosPolicy DDS_TopicQos::destination_order [read]

Destination order policy, **DESTINATION_ORDER** (p. 365).

6.139.2.9 struct DDS_HistoryQosPolicy DDS_TopicQos::history [read]

History policy, **HISTORY** (p. 367).

6.139.2.10 struct DDS_ResourceLimitsQosPolicy DDS_TopicQos::resource_limits [read]

Resource limits policy, **RESOURCE_LIMITS** (p. 371).

6.139.2.11 struct DDS_TransportPriorityQosPolicy DDS_TopicQos::transport_priority [read]

Transport priority policy, **TRANSPORT_PRIORITY** (p. 373).

6.139.2.12 struct DDS_LifespanQosPolicy DDS_TopicQos::lifespan [read]

Lifespan policy, **LIFESPAN** (p. 374).

6.139.2.13 struct DDS_OwnershipQosPolicy DDS_TopicQos::ownership [read]

Ownership policy, **OWNERSHIP** (p. 355).

6.140 DDS_TransportBuiltinQosPolicy Struct Reference

Specifies which built-in transports are used.

Public Attributes

- **DDS_TransportBuiltinKindMask mask**

  *Specifies the built-in transports that are registered automatically when the **DDSDomainParticipant** (p. 1139) is enabled.*

6.140.1 Detailed Description

Specifies which built-in transports are used.

Three different transport plug-ins are built into the core RTI Connext libraries (for most supported target platforms): UDPv4, shared memory, and UDPv6.

This QoS policy allows you to control which of these built-in transport plug-ins are used by a **DDSDomainParticipant** (p. 1139). By default, only the UDPv4 and shared memory plug-ins are enabled (although on some embedded platforms, the shared memory plug-in is not available). In some cases, users will disable the shared memory transport when they do not want applications to use shared memory to communicate when running on the same node.

Note: If one application is configured to use UDPv4 *and* shared memory, while another application is only configured for UDPv4, and these two applications run on the same node, they will not communicate. This is due to an internal optimization which will default to use shared memory instead of loopback. However if the other peer application does not enable shared memory there is no common transport, therefore they will not communicate.

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.140.2 Member Data Documentation

6.140.2.1 DDS_TransportBuiltinKindMask DDS_TransportBuiltinQosPolicy::mask

Specifies the built-in transports that are registered automatically when the **DDSDomainParticipant** (p. 1139) is enabled.

RTI Connext provides several built-in transports. Only those that are specified with this mask are registered automatically when the **DDSDomainParticipant** (p. 1139) is enabled.

[default] **DDS_TRANSPORTBUILTIN_MASK_DEFAULT** (p. 397)

6.141 DDS_TransportMulticastMapping_t Struct Reference

Type representing a list of multicast mapping elements.

Public Attributes

- char * **addresses**

  A string containing a comma-separated list of IP addresses or IP address ranges to be used to receive multicast *traffic for the entity with a topic that matches the* ***DDS_TransportMulticastMapping_t::topic_expression*** *(p. 972).*

- char * **topic_expression**

  A regular expression that will be used to map topic names to corresponding multicast receive addresses.

- struct **DDS_TransportMulticastMappingFunction_t mapping_function**

  Specifies a function that will define the mapping between a topic name and a specific multicast address from a list of addresses.

6.141.1 Detailed Description

Type representing a list of multicast mapping elements.

A multicast mapping element specifies a string containing a list of IP addresses, a topic expression and a mapping function.

QoS:

DDS_TransportMulticastMappingQosPolicy (p. 975)

6.141.2 Member Data Documentation

6.141.2.1 char* DDS_TransportMulticastMapping_t::addresses

A string containing a comma-separated list of IP addresses or IP address ranges to be used to receive *multicast* traffic for the entity with a topic that matches the **DDS_TransportMulticastMapping_t::topic_expression** (p. 972).

The string must contain IPv4 or IPv6 addresses separated by commas. For example: "239.255.100.1,239.255.100.2,239.255.100.3"

You may specify ranges of addresses by enclosing the start address and the end address in square brackets. For example: "[239.255.100.1,239.255.100.3]"

You may combine the two approaches. For example:

"239.255.200.1,[239.255.100.1,239.255.100.3], 239.255.200.3"

IPv4 addresses must be specified in Dot-decimal notation.

IPv6 addresses must be specified using 8 groups of 16-bit hexadecimal values separated by colons. For example: FF00:0000:0000:0000:0202:B3FF:FE1E:8329

Leading zeroes can be skipped. For example: "FF00:0:0:0:202:B3FF:FE1E:8329"

You may replace a consecutive number of zeroes with a double colon, but only once within an address. For example: "FF00::202:B3FF:FE1E:8329"

[default] NULL

6.141.2.2 char* DDS_TransportMulticastMapping_t::topic_expression

A regular expression that will be used to map topic names to corresponding multicast receive addresses.

A topic name must match the expression before a corresponding address is assigned.

[default] NULL

6.141.2.3 struct DDS_TransportMulticastMappingFunction_t DDS_TransportMulticastMapping_t::mapping_function [read]

Specifies a function that will define the mapping between a topic name and a specific multicast address from a list of addresses.

This function is optional. If not specified, the middleware will use a hash function to perform the mapping.

6.142 DDS_TransportMulticastMappingFunction_t Struct Reference

Type representing an external mapping function.

Public Attributes

- char * **dll**

  Specifies a dynamic library that contains a mapping function.

- char * **function_name**

  Specifies the name of a mapping function.

6.142.1 Detailed Description

Type representing an external mapping function.

A mapping function is defined by a dynamic library name and a function name.

QoS:

DDS_TransportMulticastMappingQosPolicy (p. 975)

6.142.2 Member Data Documentation

6.142.2.1 char* DDS_TransportMulticastMappingFunction_t::dll

Specifies a dynamic library that contains a mapping function.

A relative or absolute path can be specified.

If the name is specified as "foo", the library name on Linux systems will be libfoo.so; on Windows systems it will be foo.dll.

[**default**] NULL

6.142.2.2 char* DDS_TransportMulticastMappingFunction_t::function_name

Specifies the name of a mapping function.

This function must be implemented in the library specified in **DDS_TransportMulticastMappingFunction_t::dll** (p. 973).

The function must implement the following interface:

int function(const char* topic_name, int numberOfAddresses);

The function must return an integer that indicates the *index* of the address to use for the given `topic_name`. For example, if the first address in the list should be used, it must return 0; if the second address in the list should be used, it must return 1, etc.

6.143 DDS_TransportMulticastMappingQosPolicy Struct Reference

Specifies a list of topic_expressions and multicast addresses that can be used by an Entity with a specific topic name to receive data.

Public Attributes

- struct **DDS_TransportMulticastMappingSeq value**
  A sequence of multicast communication mappings.

6.143.1 Detailed Description

Specifies a list of topic_expressions and multicast addresses that can be used by an Entity with a specific topic name to receive data.

This QoS policy provides an alternate way to assign multicast receive addresses to DataReaders. It allows you to perform multicast configuration at the **DDSDomainParticipant** (p. 1139) level.

To use multicast communication *without* this QoS policy, you must explicitly assign a multicast receive address on each **DDSDataReader** (p. 1087). This can quickly become difficult to configure as more DataReaders of different topics and multicast addresses are added.

With this QoS policy, you can configure a set of multicast addresses on the **DDSDomainParticipant** (p. 1139); those addresses will then be automatically assigned to the DomainParticipant's DataReaders. A single configuration on the **DDSDomainParticipant** (p. 1139) can thus replace per-DataReader configuration.

On the DomainParticipant, the set of assignable addresses can be configured for specific topics. Addresses are configured on topics because efficient usage of multicast will have all DataWriters and DataReaders of a single topic using the same multicast address.

You can specify a mapping between a topic's name and a multicast address. For example, topic 'A' can be assigned to address 239.255.1.1 and topic 'B' can be assigned to address 239.255.1.2.

You can use filter expressions to configure a subset of topics to use a specific list of addresses. For example, suppose topics "X", "Y" and "Z" need to be sent to any address within the range [239.255.1.1, 239.255.1.255]. You can specify an expression on the topic name (e.g. "[X-Z]") corresponding to that range of addresses. Then the DomainParticipant will select an address for a topic whose name matches the expression.

The middleware will use a hash function to perform the mapping from topic to address. Alternatively, you can specify a pluggable mapping function.

IMPORTANT: All the strings defined in each element of the sequence must be assigned using RTI_String_dup("foo");. For example:

mcastMappingElement->addresses = DDS_String_dup("[239.255.1.1,239.255.1.255]");

NOTE: To use this QoS policy, you must set **DDS_TransportMulticastQosPolicy::kind** (p. 979) to **DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS** (p. 385).

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.143.2 Member Data Documentation

6.143.2.1 struct DDS_TransportMulticastMappingSeq DDS_TransportMulticastMappingQosPolicy::value [read]

A sequence of multicast communication mappings.

[default] Empty sequence.

6.144 DDS_TransportMulticastMappingSeq Struct Reference

Declares IDL `sequence<` **DDS_TransportMulticastMapping_t** (p. 971) `>`.

6.144.1 Detailed Description

Declares IDL `sequence<` **DDS_TransportMulticastMapping_t** (p. 971) `>`.

Instantiates:

<<generic>> (p. *199*) **FooSeq** (p. 1494)

See also:

DDS_TransportMulticastMapping_t (p. 971)

6.145 DDS_TransportMulticastQosPolicy Struct Reference

Specifies the multicast address on which a **DDSDataReader** (p. 1087) wants to receive its data. It can also specify a port number as well as a subset of the available (at the **DDSDomainParticipant** (p. 1139) level) transports with which to receive the multicast data.

Public Attributes

- struct **DDS_TransportMulticastSettingsSeq value**

  A sequence of multicast communications settings.

- **DDS_TransportMulticastQosPolicyKind kind**

  A value that specifies a way to determine how to obtain the multicast address.

6.145.1 Detailed Description

Specifies the multicast address on which a **DDSDataReader** (p. 1087) wants to receive its data. It can also specify a port number as well as a subset of the available (at the **DDSDomainParticipant** (p. 1139) level) transports with which to receive the multicast data.

By default, a **DDSDataWriter** (p. 1113) will send individually addressed packets for each **DDSDataReader** (p. 1087) that subscribes to the topic of the DataWriter – this is known as unicast delivery. Thus, as many copies of the data will be sent over the network as there are DataReaders for the data. The network bandwidth used by a DataWriter will thus increase linearly with the number of DataReaders.

Multicast addressing (on UDP/IP transports) allows multiple DataReaders to receive the *same* network packet. By using multicast, a **DDSDataWriter** (p. 1113) can send a single network packet that is received by all subscribing applications. Thus the network bandwidth usage will be constant, independent of the number of DataReaders.

Coordinating the multicast address specified by DataReaders can help optimize network bandwidth usage in systems where there are multiple DataReaders for the same **DDSTopic** (p. 1419).

Entity:

DDSDataReader (p. 1087)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.145.2 Member Data Documentation

6.145.2.1 struct DDS_TransportMulticastSettingsSeq DDS_TransportMulticastQosPolicy::value [read]

A sequence of multicast communications settings.

An empty sequence means that multicast is not used by the entity.

The RTPS wire protocol currently limits the maximum number of multicast locators to four.

[**default**] Empty sequence.

6.145.2.2 DDS_TransportMulticastQosPolicyKind DDS_TransportMulticastQosPolicy::kind

A value that specifies a way to determine how to obtain the multicast address.

This field can have two values.

- If it is set to **DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS** (p. 385) and the **DDS_TransportMulticastQosPolicy::value** (p. 979) does not have any elements, then multicast will not be used.
- If it is set to **DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS** (p. 385) and the **DDS_TransportMulticastQosPolicy::value** (p. 979) has at least one element with a valid address, then that address will be used.
- If it is set to **DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS** (p. 385) and the **DDS_TransportMulticastQosPolicy::value** (p. 979) has at least one element with an empty address, then the address will be obtained from **DDS_TransportMulticastMappingQosPolicy** (p. 975).
- If it is set to **DDS_UNICAST_ONLY_TRANSPORT_MULTICAST_QOS** (p. 385), then multicast will not be used.

[**default**] **DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS** (p. 385)

6.146 DDS_TransportMulticastSettings_t Struct Reference

Type representing a list of multicast locators.

Public Attributes

- struct **DDS_StringSeq transports**

  A sequence of transport aliases that specifies the transports on which to receive multicast *traffic for the entity.*

- char * **receive_address**

  The multicast group address on which the entity can receive data.

- **DDS_Long receive_port**

  The multicast port on which the entity can receive data.

6.146.1 Detailed Description

Type representing a list of multicast locators.

A multicast locator specifies a transport class, a multicast address, and a multicast port number on which messages can be received by an entity.

QoS:

DDS_TransportMulticastQosPolicy (p. 978)

6.146.2 Member Data Documentation

6.146.2.1 struct DDS_StringSeq DDS_TransportMulticastSettings_t::transports [read]

A sequence of transport aliases that specifies the transports on which to receive *multicast* traffic for the entity.

Of the transport instances available to the entity, only those with aliases matching an alias in this sequence are used to subscribe to the multicast group addresses. Thus, this list of aliases sub-selects from the transport s available to the entity.

An empty sequence is a special value that specifies all the transports available to the entity.

The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

Alias names for the builtin transports are defined in **TRANSPORT_BUILTIN** (p. 396).

[**default**] Empty sequence; i.e. all the transports available to the entity.

[**range**] Any sequence of non-null, non-empty strings.

6.146.2.2 char* DDS_TransportMulticastSettings_t::receive_address

The multicast group address on which the entity can receive data.

Must must be an address in the proper format (see **Address Format** (p. 390)).

[**default**] NONE/INVALID. Required to specify a multicast group address to join.

[**range**] A valid IPv4 or IPv6 multicast address.

See also:

Address Format (p. 390)

6.146.2.3 DDS_Long DDS_TransportMulticastSettings_t::receive_port

The multicast port on which the entity can receive data.

[**default**] 0, which implies that the actual port number is determined by a formula as a function of the `domain_id` (see **DDS_WireProtocolQosPolicy::participant_id** (p. 1063)).

[**range**] [0,0xffffffff]

6.147 DDS_TransportMulticastSettingsSeq Struct Reference

Declares IDL `sequence<` **DDS_TransportMulticastSettings_t** (p. 980) `>`.

6.147.1 Detailed Description

Declares IDL `sequence<` **DDS_TransportMulticastSettings_t** (p. 980) `>`.

Instantiates:

<<generic>> (p. *199*) **FooSeq** (p. 1494)

See also:

DDS_TransportMulticastSettings_t (p. 980)

6.148 DDS_TransportPriorityQosPolicy Struct Reference

This QoS policy allows the application to take advantage of transports that are capable of sending messages with different priorities.

Public Attributes

- **DDS_Long value**

  This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.

6.148.1 Detailed Description

This QoS policy allows the application to take advantage of transports that are capable of sending messages with different priorities.

The Transport Priority QoS policy is optional and only supported on certain OSs and transports. It allows you to specify on a per-**DDSDataWriter** (p. 1113) basis that the data sent by that **DDSDataWriter** (p. 1113) is of a different priority.

The DDS specification does not indicate how a DDS implementation should treat data of different priorities. It is often difficult or impossible for DDS implementations to treat data of higher priority differently than data of lower priority, especially when data is being sent (delivered to a physical transport) directly by the thread that called **FooDataWriter::write** (p. 1484). Also, many physical network transports themselves do not have a end-user controllable level of data packet priority.

Entity:

DDSDataWriter (p. 1113), **DDSTopic** (p. 1419)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **YES** (p. 340)

6.148.2 Usage

In RTI Connext, for the ::**UDPv4 Transport** (p. 265), the value set in the Transport Priority QoS policy is used in a `setsockopt` call to set the TOS (type

of service) bits of the IPv4 header for datagrams sent by a **DDSDataWriter** (p. 1113). It is platform-dependent how and whether the `setsockopt` has an effect. On some platforms, such as Windows and Linux, external permissions must be given to the user application in order to set the TOS bits.

It is incorrect to assume that using the Transport Priority QoS policy will have any effect at all on the end-to-end delivery of data from a **DDSDataWriter** (p. 1113) to a **DDSDataReader** (p. 1087). All network elements, including switches and routers must have the capability and be enabled to actually use the TOS bits to treat higher priority packets differently. Thus the ability to use the Transport Priority QoS policy must be designed and configured at a *system* level; just turning it on in an application may have no effect at all.

6.148.3 Member Data Documentation

6.148.3.1 DDS_Long DDS_TransportPriorityQosPolicy::value

This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.

You may choose any value within the range of a 32-bit signed integer; higher values indicate higher priority. However, any further interpretation of this policy is specific to a particular transport and a particular DDS implementation. For example, a particular transport is permitted to treat a range of priority values as equivalent to one another.

[default] 0

6.149 DDS_TransportSelectionQosPolicy Struct Reference

Specifies the physical transports a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087) may use to send or receive data.

Public Attributes

- struct **DDS_StringSeq enabled_transports**

  A sequence of transport aliases that specifies the transport instances available for use by the entity.

6.149.1 Detailed Description

Specifies the physical transports a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087) may use to send or receive data.

An application may be simultaneously connected to many different physical transports, e.g., Ethernet, Infiniband, shared memory, VME backplane, and wireless. By default, RTI Connext will use up to 4 transports to deliver data from a DataWriter to a DataReader.

This QoS policy can be used to both limit and control which of the application's available transports may be used by a **DDSDataWriter** (p. 1113) to send data or by a **DDSDataReader** (p. 1087) to receive data.

Entity:

DDSDataReader (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.149.2 Member Data Documentation

6.149.2.1 struct DDS_StringSeq DDS_TransportSelectionQosPolicy::enabled_transports [read]

A sequence of transport aliases that specifies the transport instances available for use by the entity.

Of the transport instances installed with the **DDSDomainParticipant** (p. 1139), only those with aliases matching an alias in this sequence are available to the entity.

Thus, this list of aliases sub-selects from the transports available to the **DDSDomainParticipant** (p. 1139).

An empty sequence is a special value that specifies all the transports installed with the **DDSDomainParticipant** (p. 1139).

The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

Alias names for the builtin transports are defined in **TRANSPORT_BUILTIN** (p. 396).

[**default**] Empty sequence; i.e. all the transports installed with and available to the **DDSDomainParticipant** (p. 1139).

[**range**] A sequence of non-null, non-empty strings.

See also:

DDS_DomainParticipantQos::transport_builtin (p. 590).

6.150 DDS_TransportUnicastQosPolicy Struct Reference

Specifies a subset of transports and a port number that can be used by an Entity to receive data.

Public Attributes

- struct **DDS_TransportUnicastSettingsSeq value**

  A sequence of unicast communication settings.

6.150.1 Detailed Description

Specifies a subset of transports and a port number that can be used by an Entity to receive data.

Entity:

DDSDomainParticipant (p. 1139), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.150.2 Usage

RTI Connext may send data to a variety of Entities, not just DataReaders. For example, reliable DataWriters may receive ACK/NACK packets from reliable DataReaders.

During discovery, each **DDSEntity** (p. 1253) announces to remote applications a list of (up to 4) unicast addresses to which the remote application should send data (either user data packets or reliable protocol meta-data such as ACK/NACKs and heartbeats).

By default, the list of addresses is populated automatically with values obtained from the enabled transport plug-ins allowed to be used by the Entity (see **DDS_TransportBuiltinQosPolicy** (p. 969) and **DDS_TransportSelectionQosPolicy** (p. 985)). Also, the associated ports are automatically determined (see **DDS_RtpsWellKnownPorts_t** (p. 905)).

Use this QoS policy to manually set the receive address list for an Entity. You may optionally set a port to use a non-default receive port as well. Only the first 4 addresses will be used.

RTI Connext will create a receive thread for every unique port number that it encounters (on a per transport basis).

- For a **DDSDomainParticipant** (p. 1139), this QoS policy sets the default list of addresses used by other applications to send user data for local DataReaders.

- For a **DDSDataReader** (p. 1087), if set, then other applications will use the specified list of addresses to send user data (and reliable protocol packets for reliable DataReaders). Otherwise, if not set, the other applications will use the addresses set by the **DDSDomainParticipant** (p. 1139).

- For a reliable **DDSDataWriter** (p. 1113), if set, then other applications will use the specified list of addresses to send reliable protocol packets (ACKS/NACKS) on the behalf of reliable DataReaders. Otherwise, if not set, the other applications will use the addresses set by the **DDSDomainParticipant** (p. 1139).

6.150.3 Member Data Documentation

6.150.3.1 struct DDS_TransportUnicastSettingsSeq DDS_TransportUnicastQosPolicy::value [read]

A sequence of unicast communication settings.

An empty sequence means that applicable defaults specified by elsewhere (e.g. **DDS_DomainParticipantQos::default_unicast** (p. 591)) should be used.

The RTPS wire protocol currently limits the maximum number of unicast locators to four.

[default] Empty sequence.

See also:

DDS_DomainParticipantQos::default_unicast (p. 591)

6.151 DDS_TransportUnicastSettings_t Struct Reference

Type representing a list of unicast locators.

Public Attributes

- struct **DDS_StringSeq transports**

  A sequence of transport aliases that specifies the unicast interfaces on which to receive unicast *traffic for the entity.*

- **DDS_Long receive_port**

  The unicast port on which the entity can receive data.

6.151.1 Detailed Description

Type representing a list of unicast locators.

A unicast locator specifies a transport class, a unicast address, and a unicast port number on which messages can be received by an entity.

QoS:

DDS_TransportUnicastQosPolicy (p. 987)

6.151.2 Member Data Documentation

6.151.2.1 struct DDS_StringSeq DDS_TransportUnicastSettings_t::transports [read]

A sequence of transport aliases that specifies the unicast interfaces on which to receive *unicast* traffic for the entity.

Of the transport instances available to the entity, only those with aliases matching an alias on this sequence are used to determine the unicast interfaces used by the entity.

Thus, this list of aliases sub-selects from the transports available to the entity.

Each unicast interface on a transport results in a unicast locator for the entity.

An empty sequence is a special value that specifies all the transports available to the entity.

The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

Alias names for the builtin transports are defined in **TRANSPORT_BUILTIN** (p. 396).

[**default**] Empty sequence; i.e. all the transports available to the entity.

[**range**] Any sequence of non-null, non-empty strings.

6.151.2.2 DDS_Long DDS_TransportUnicastSettings_t::receive_port

The unicast port on which the entity can receive data.

Must be an *unused* unicast port on the system.

[**default**] 0, which implies that the actual port number is determined by a formula as a function of the `domain_id`, and the **DDS_WireProtocolQosPolicy::participant_id** (p. 1063).

[**range**] [0,0xffffffff]

See also:

DDS_WireProtocolQosPolicy::participant_id (p. 1063).

6.152 DDS_TransportUnicastSettingsSeq Struct Reference

Declares IDL `sequence<` **DDS_TransportUnicastSettings_t** (p. 989) `>`.

6.152.1 Detailed Description

Declares IDL `sequence<` **DDS_TransportUnicastSettings_t** (p. 989) `>`.

Instantiates:

<<generic>> (p. *199*) **FooSeq** (p. 1494)

See also:

DDS_TransportUnicastSettings_t (p. 989)

6.153 DDS_TypeCode Struct Reference

The definition of a particular data type, which you can use to inspect the name, members, and other properties of types generated with **rtiddsgen** (p. 220) or to modify types you define yourself at runtime.

Public Member Functions

- **DDS_TCKind kind** (**DDS_ExceptionCode_t** &ex) const

  *Gets the **DDS_TCKind** (p. 66) value of a type code.*

- **DDS_ExtensibilityKind extensibility_kind** (**DDS_ExceptionCode_t** &ex) const

  *Gets the **DDS_ExtensibilityKind** (p. 67) value of a type code.*

- **DDS_Boolean equal** (const **DDS_TypeCode** *tc, **DDS_ExceptionCode_t** &ex) const

  *Compares two **DDS_TypeCode** (p. 992) objects for equality.*

- const char * **name** (**DDS_ExceptionCode_t** &ex) const

  *Retrieves the simple name identifying this **DDS_TypeCode** (p. 992) object within its enclosing scope.*

- **DDS_UnsignedLong member_count** (**DDS_ExceptionCode_t** &ex) const

  Returns the number of members of the type code.

- const char * **member_name** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Returns the name of a type code member identified by the given index.

- **DDS_UnsignedLong find_member_by_name** (const char *name, **DDS_ExceptionCode_t** &ex) const

  Get the index of the member of the given name.

- **DDS_TypeCode** * **member_type** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  *Retrieves the **DDS_TypeCode** (p. 992) object describing the type of the member identified by the given index.*

- **DDS_UnsignedLong member_label_count** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Returns the number of labels associated to the index-th union member.

- **DDS_Long member_label** (**DDS_UnsignedLong** member_index, **DDS_UnsignedLong** label_index, **DDS_ExceptionCode_t** &ex) const

  Return the label_index-th label associated to the member_index-th member.

- **DDS_Long member_ordinal** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Returns the ordinal that corresponds to the index-th enum value.

- **DDS_Boolean is_member_key** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Function that tells if a member is a key or not.

- **DDS_Boolean is_member_required** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Indicates whether a given member of a type is required to be present in every sample of that type.

- **DDS_Boolean is_member_pointer** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Function that tells if a member is a pointer or not.

- **DDS_Boolean is_member_bitfield** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Function that tells if a member is a bitfield or not.

- **DDS_Short member_bitfield_bits** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Returns the number of bits of a bitfield member.

- **DDS_Visibility member_visibility** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

  Returns the constant that indicates the visibility of the index-th member.

- **DDS_TypeCode** * **discriminator_type** (**DDS_ExceptionCode_t** &ex) const

  Returns the discriminator type code.

- **DDS_UnsignedLong length** (**DDS_ExceptionCode_t** &ex) const

  Returns the number of elements in the type described by this type code.

- **DDS_UnsignedLong array_dimension_count** (**DDS_ExceptionCode_t** &ex) const

This function returns the number of dimensions of an array type code.

^ **DDS_UnsignedLong array_dimension** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

This function returns the index-th dimension of an array type code.

^ **DDS_UnsignedLong element_count** (**DDS_ExceptionCode_t** &ex) const

The number of elements in an array.

^ **DDS_TypeCode** * **content_type** (**DDS_ExceptionCode_t** &ex) const

*Returns the **DDS_TypeCode** (p. 992) object representing the type for the members of the object described by this **DDS_TypeCode** (p. 992) object.*

^ **DDS_Boolean is_alias_pointer** (**DDS_ExceptionCode_t** &ex) const

Function that tells if an alias is a pointer or not.

^ **DDS_Long default_index** (**DDS_ExceptionCode_t** &ex) const

Returns the index of the default member, or -1 if there is no default member.

^ **DDS_TypeCode** * **concrete_base_type** (**DDS_ExceptionCode_t** &ex) const

*Returns the **DDS_TypeCode** (p. 992) that describes the concrete base type of the value type that this **DDS_TypeCode** (p. 992) object describes.*

^ **DDS_ValueModifier type_modifier** (**DDS_ExceptionCode_t** &ex) const

*Returns a constant indicating the modifier of the value type that this **DDS_TypeCode** (p. 992) object describes.*

^ **DDS_Long member_id** (**DDS_UnsignedLong** index, **DDS_ExceptionCode_t** &ex) const

Returns the ID of a sparse type code member identified by the given index.

^ **DDS_UnsignedLong find_member_by_id** (**DDS_Long** id, **DDS_ExceptionCode_t** &ex) const

Get the index of the member of the given ID.

^ **DDS_UnsignedLong add_member_to_enum** (const char *name, **DDS_Long** ordinal, **DDS_ExceptionCode_t** &ex)

*Add a new enumerated constant to this enum **DDS_TypeCode** (p. 992).*

- **DDS_UnsignedLong add_member** (const char *name, **DDS_Long** id, const **DDS_TypeCode** *tc, **DDS_Octet** member_flags, **DDS_ExceptionCode_t** &ex)

  *Add a new member to this **DDS_TypeCode** (p. 992).*

- **DDS_UnsignedLong add_member_ex** (const char *name, **DDS_Long** id, const **DDS_TypeCode** *tc, **DDS_Octet** member_flags, **DDS_Visibility** visibility, **DDS_Boolean** is_pointer, **DDS_Short** bits, **DDS_ExceptionCode_t** &ex)

  *Add a new member to this **DDS_TypeCode** (p. 992).*

- void **print_IDL** (**DDS_UnsignedLong** indent, **DDS_ExceptionCode_t** &ex) const

  *Prints a **DDS_TypeCode** (p. 992) in a pseudo-IDL notation.*

6.153.1 Detailed Description

The definition of a particular data type, which you can use to inspect the name, members, and other properties of types generated with **rtiddsgen** (p. 220) or to modify types you define yourself at runtime.

You create **DDS_TypeCode** (p. 992) objects using the **DDS_TypeCodeFactory** (p. 1022) singleton. Then you can use the methods on *this* class to inspect and modify the data type definition.

This class is based on a similar class from CORBA.

MT Safety:

SAFE for read-only access, UNSAFE for modification. Modifying a single **DDS_TypeCode** (p. 992) object concurrently from multiple threads is *unsafe*. Modifying a **DDS_TypeCode** (p. 992) from a single thread while concurrently reading the state of that **DDS_TypeCode** (p. 992) from another thread is also *unsafe*. However, reading the state of a **DDS_TypeCode** (p. 992) concurrently from multiple threads, without any modification, is *safe*.

Examples:

HelloWorld.cxx.

6.153.2 Member Function Documentation

6.153.2.1 DDS_TCKind DDS_TypeCode::kind (DDS_ExceptionCode_t & *ex*) const

Gets the **DDS_TCKind** (p. 66) value of a type code.

Parameters:

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that it can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Retrieves the kind of this **DDS_TypeCode** (p. 992) object. The kind of a type code determines which **DDS_TypeCode** (p. 992) methods may legally be invoked on it.

MT Safety:

SAFE.

Returns:

The type code kind.

6.153.2.2 DDS_ExtensibilityKind DDS_TypeCode::extensibility_kind (DDS_ExceptionCode_t & *ex*) const

Gets the **DDS_ExtensibilityKind** (p. 67) value of a type code.

Parameters:

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that it can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Retrieves the extensibility kind of this **DDS_TypeCode** (p. 992) object.

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. 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.

In order to support use cases such as these, the type system introduces the concept of extensible and mutable types.

- A type may be final, indicating that the range of its possible data values is strictly defined. In particular, it is not possible to add elements to members of collection or aggregated types while maintaining type assignability.
- A type may be extensible, indicating that two types, where one contains all of the elements/members of the other plus additional elements/members appended to the end, may remain assignable.
- A type may be mutable, indicating that two types may differ from one another in the additional, removal, and/or transposition of elements/members while remaining assignable.

The extensibility of **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_VALUE** (p. 67), and **DDS_TK_ENUM** (p. 66) can be change using the built-in Extensibility annotation when the type is declared.

IDL example:

```
struct MyType {
     long member_1;
 } //@Extensibility EXTENSIBLE_EXTENSIBILITY
```

XML example:

```
<struct name="MyType" extensibility="extensible">
     <member name="member_1" type="long"/>
 </struct>
```

XSD example:

```
<xsd:complexType name="MyType">
     <xsd:sequence>
         <xsd:element name="member_1" minOccurs="1" maxOccurs="1" type="xsd:int"/>
     </xsd:sequence>
 </xsd:complexType>
 <!-- @struct true -->
 <!-- @extensibility EXTENSIBLE_EXTENSIBILITY -->
```

For TypeCodes built at run-time using the **DDS_TypeCodeFactory** (p. 1022) API, the extensibility can be provided as a parameter of the following APIs:

- **DDS_TypeCodeFactory::create_struct_tc** (p. 1027)
- **DDS_TypeCodeFactory::create_value_tc** (p. 1028)
- **DDS_TypeCodeFactory::create_union_tc** (p. 1030)
- **DDS_TypeCodeFactory::create_enum_tc** (p. 1032)

See also:

DDS_ExtensibilityKind (p. 67)

MT Safety:

SAFE.

Returns:

The type code extensibility kind.

6.153.2.3 DDS_Boolean DDS_TypeCode::equal (const DDS_TypeCode * *tc*, DDS_ExceptionCode_t & *ex*) const

Compares two **DDS_TypeCode** (p. 992) objects for equality.

MT Safety:

SAFE.

For equality and assignability purposes, **DDS_TK_STRUCT** (p. 66) and **DDS_TK_VALUE** (p. 67) are considered equivalent.

The **DDS_TypeCode** (p. 992) of structs inheriting from other structs has a **DDS_TK_VALUE** (p. 67) kind.

For example:

```
struct MyStruct: MyBaseStruct {
     long member_1;
 };
```

The code generation for the previous type will generate a **DDS_TypeCode** (p. 992) with **DDS_TK_VALUE** (p. 67) kind.

Parameters:

tc <<*in*>> *(p. 200)* Type code that will be compared with this **DDS_TypeCode** (p. 992).

ex <<*out*>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the type codes are equal. Otherwise, **DDS_BOOLEAN_FALSE** (p. 299).

6.153.2.4 const char* DDS_TypeCode::name (DDS_ExceptionCode_t & *ex*) const

Retrieves the simple name identifying this **DDS_TypeCode** (p. 992) object within its enclosing scope.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_ENUM** (p. 66), **DDS_TK_VALUE** (p. 67), **DDS_TK_SPARSE** (p. 67) or **DDS_TK_ALIAS** (p. 66).

MT Safety:

SAFE.

Parameters:

ex <<*out*>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)

- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

Name of the type code if no errors.

6.153.2.5 DDS_UnsignedLong DDS_TypeCode::member_count (DDS_ExceptionCode_t & *ex*) const

Returns the number of members of the type code.

The method member_count can be invoked on structure, union, and enumeration **DDS_TypeCode** (p. 992) objects.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_ENUM** (p. 66), **DDS_TK_VALUE** (p. 67) or **DDS_TK_SPARSE** (p. 67).

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

The number of members constituting the type described by this **DDS_TypeCode** (p. 992) object if no errors.

6.153.2.6 const char* DDS_TypeCode::member_name (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Returns the name of a type code member identified by the given index.

The method member_name can be invoked on structure, union, and enumeration **DDS_TypeCode** (p. 992) objects.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_ENUM** (p. 66), **DDS_TK_VALUE** (p. 67) or **DDS_TK_SPARSE** (p. 67).
The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index <<***in***>> *(p. 200)* Member index in the interval [0,(member count-1)].

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

Name of the member if no errors.

6.153.2.7 DDS_UnsignedLong DDS_TypeCode::find_member_by_name (const char * *name*, DDS_ExceptionCode_t & *ex*) const

Get the index of the member of the given name.

MT Safety:

SAFE.

6.153.2.8 DDS_TypeCode* DDS_TypeCode::member_type (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Retrieves the **DDS_TypeCode** (p. 992) object describing the type of the member identified by the given index.

The method member_type can be invoked on structure and union type codes.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66), **DDS_TK_VALUE** (p. 67) or **DDS_TK_SPARSE** (p. 67).
The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index *<<**in**>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

The **DDS_TypeCode** (p. 992) object describing the member at the given index if no errors.

6.153.2.9 DDS_UnsignedLong DDS_TypeCode::member_label_count (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Returns the number of labels associated to the index-th union member.

The method can be invoked on union **DDS_TypeCode** (p. 992) objects.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_UNION** (p. 66).
The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index *<<**in**>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

Number of labels if no errors.

6.153.2.10 DDS_Long DDS_TypeCode::member_label (DDS_UnsignedLong *member_index*, DDS_UnsignedLong *label_index*, DDS_ExceptionCode_t & *ex*) const

Return the label_index-th label associated to the member_index-th member.

This method has been modified for RTI Connext from the CORBA Type code Specification.

Example:

case 1: Label index 0

case 2: Label index 1

short short_member;

The method can be invoked on union **DDS_TypeCode** (p. 992) objects.

Precondition:

self kind is **DDS_TK_UNION** (p. 66).
The member_index param must be in the interval [0,(member count-1)].
The label_index param must be in the interval [0,(member labels count-1)].

MT Safety:

SAFE.

Parameters:

member_index *<<**in**>>* *(p. 200)* Member index.

label_index *<<**in**>>* *(p. 200)* Label index.

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

The evaluated value of the label if no errors.

6.153.2.11 DDS_Long DDS_TypeCode::member_ordinal (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Returns the ordinal that corresponds to the index-th enum value.

The method can be invoked on enum **DDS_TypeCode** (p. 992) objects.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_ENUM** (p. 66).
Member index in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index *<<**in**>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

Ordinal that corresponds to the index-th enumerator if no errors.

6.153.2.12 DDS_Boolean DDS_TypeCode::is_member_key (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Function that tells if a member is a key or not.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66), **DDS_TK_VALUE** (p. 67) or **DDS_TK_SPARSE** (p. 67).
The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index <<*in*>> *(p. 200)* Member index in the interval [0,(member count-1)].

ex <<*out*>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the member is a key. Otherwise, **DDS_BOOLEAN_FALSE** (p. 299).

6.153.2.13 DDS_Boolean DDS_TypeCode::is_member_required (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Indicates whether a given member of a type is required to be present in every sample of that type.

Which fields are required depends on the **DDS_TCKind** (p. 66) of the type. For example, in a type of kind **DDS_TK_SPARSE** (p. 67), key fields are required. In **DDS_TK_STRUCT** (p. 66) and **DDS_TK_VALUE** (p. 67) types, all fields are required.

MT Safety:

SAFE.

6.153.2.14 DDS_Boolean DDS_TypeCode::is_member_pointer (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Function that tells if a member is a pointer or not.

The method is_member_pointer can be invoked on union and structs type objects

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66), **DDS_TK_UNION** (p. 66) or **DDS_TK_VALUE** (p. 67).
The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index <<***in***>> *(p. 200)* Index of the member for which type information is begin requested.

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the member is a pointer. Otherwise, **DDS_BOOLEAN_FALSE** (p. 299).

6.153.2.15 DDS_Boolean DDS_TypeCode::is_member_bitfield (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Function that tells if a member is a bitfield or not.

The method can be invoked on struct type objects.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66) or **DDS_TK_VALUE** (p. 67). The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index *<<in>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<out>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the member is a bitfield. Otherwise, **DDS_BOOLEAN_FALSE** (p. 299).

6.153.2.16 DDS_Short DDS_TypeCode::member_bitfield_bits (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Returns the number of bits of a bitfield member.

The method can be invoked on struct type objects.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_STRUCT** (p. 66) or **DDS_TK_VALUE** (p. 67). The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index *<<**in**>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

The number of bits of the bitfield or DDS_TypeCode::NOT_BITFIELD if the member is not a bitfield.

6.153.2.17 DDS_Visibility DDS_TypeCode::member_visibility (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Returns the constant that indicates the visibility of the index-th member.

Precondition:

self kind is **DDS_TK_VALUE** (p. 67), **DDS_TK_STRUCT** (p. 66) or **DDS_TK_SPARSE** (p. 67). The index param must be in the interval [0,(member count-1)].

MT Safety:

SAFE.

For **DDS_TK_STRUCT** (p. 66), this method always returns **DDS_PUBLIC_MEMBER** (p. 63).

Parameters:

index *<<**in**>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

One of the following constants: **DDS_PRIVATE_MEMBER** (p. 63) or **DDS_PUBLIC_MEMBER** (p. 63).

6.153.2.18 DDS_TypeCode* DDS_TypeCode::discriminator_type (DDS_ExceptionCode_t & *ex*) const

Returns the discriminator type code.

The method discriminator_type can be invoked only on union **DDS_TypeCode** (p. 992) objects.

Precondition:

self kind is **DDS_TK_UNION** (p. 66).

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

DDS_TypeCode (p. 992) object describing the discriminator of the union type if no errors.

6.153.2.19 DDS_UnsignedLong DDS_TypeCode::length (DDS_ExceptionCode_t & *ex*) const

Returns the number of elements in the type described by this type code.

Length is:

- The maximum length of the string for string type codes.
- The maximum length of the sequence for sequence type codes.
- The first dimension of the array for array type codes.

Precondition:

self kind is **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66), **DDS_TK_STRING** (p. 66) or **DDS_TK_WSTRING** (p. 67).

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

The bound for strings and sequences, or the number of elements for arrays if no errors.

6.153.2.20 DDS_UnsignedLong DDS_TypeCode::array_dimension_count (DDS_ExceptionCode_t & *ex*) const

This function returns the number of dimensions of an array type code.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_ARRAY** (p. 66).

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

Number of dimensions if no errors.

6.153.2.21 DDS_UnsignedLong DDS_TypeCode::array_dimension (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

This function returns the index-th dimension of an array type code.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_ARRAY** (p. 66).
Dimension index in the interval [0,(dimensions count-1)].

MT Safety:

SAFE.

Parameters:

index <<***in***>> *(p. 200)* Dimension index in the interval [0,(dimensions count-1)].

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

Requested dimension if no errors.

6.153.2.22 DDS_UnsignedLong DDS_TypeCode::element_count (DDS_ExceptionCode_t & *ex*) const

The number of elements in an array.

This operation isn't relevant for other kinds of types.

MT Safety:

SAFE.

6.153.2.23 DDS_TypeCode* DDS_TypeCode::content_type (DDS_ExceptionCode_t & *ex*) const

Returns the **DDS_TypeCode** (p. 992) object representing the type for the members of the object described by this **DDS_TypeCode** (p. 992) object.

For sequences and arrays, it returns the element type. For aliases, it returns the original type.

Precondition:

self kind is **DDS_TK_ARRAY** (p. 66), **DDS_TK_SEQUENCE** (p. 66) or **DDS_TK_ALIAS** (p. 66).

MT Safety:

SAFE.

Parameters:

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

A **DDS_TypeCode** (p. 992) object representing the element type for sequences and arrays, and the original type for aliases.

6.153.2.24 DDS_Boolean DDS_TypeCode::is_alias_pointer (DDS_ExceptionCode_t & *ex*) const

Function that tells if an alias is a pointer or not.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_ALIAS** (p. 66).

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

DDS_BOOLEAN_TRUE (p. 298) if an alias is a pointer to the aliased type. Otherwise, **DDS_BOOLEAN_FALSE** (p. 299).

6.153.2.25 DDS_Long DDS_TypeCode::default_index (DDS_ExceptionCode_t & *ex*) const

Returns the index of the default member, or -1 if there is no default member.

The method default_index can be invoked only on union **DDS_TypeCode** (p. 992) objects.

Precondition:

self kind is **DDS_TK_UNION** (p. 66)

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

The index of the default member, or -1 if there is no default member.

6.153.2.26 DDS_TypeCode* DDS_TypeCode::concrete_base_type (DDS_ExceptionCode_t & *ex*) const

Returns the **DDS_TypeCode** (p. 992) that describes the concrete base type of the value type that this **DDS_TypeCode** (p. 992) object describes.

Precondition:

self kind is **DDS_TK_VALUE** (p. 67), **DDS_TK_STRUCT** (p. 66), or **DDS_TK_SPARSE** (p. 67).

MT Safety:

SAFE.

For **DDS_TK_STRUCT** (p. 66), this method always returns **DDS_TK_NULL** (p. 66).

The **DDS_TypeCode** (p. 992) of structs inheriting from other structs has a **DDS_TK_VALUE** (p. 67) kind.

For example:

```
struct MyStruct: MyBaseStruct {
     long member_1;
 };
```

The code generation for the previous type will generate a **DDS_TypeCode** (p. 992) with **DDS_TK_VALUE** (p. 67) kind.

Parameters:

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

DDS_TypeCode (p. 992) that describes the concrete base type or NULL if there is no a concrete base type.

6.153.2.27 DDS_ValueModifier DDS_TypeCode::type_modifier (DDS_ExceptionCode_t & *ex*) const

Returns a constant indicating the modifier of the value type that this **DDS_TypeCode** (p. 992) object describes.

Precondition:

self kind is **DDS_TK_VALUE** (p. 67), **DDS_TK_STRUCT** (p. 66), or **DDS_TK_SPARSE** (p. 67).

For **DDS_TK_STRUCT** (p. 66), this method always returns **DDS_VM_NONE** (p. 62).

MT Safety:

SAFE.

Parameters:

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

Returns:

One of the following type modifiers: **DDS_VM_NONE** (p. 62), **DDS_VM_ABSTRACT** (p. 63), **DDS_VM_CUSTOM** (p. 62) or **DDS_VM_TRUNCATABLE** (p. 63).

6.153.2.28 DDS_Long DDS_TypeCode::member_id (DDS_UnsignedLong *index*, DDS_ExceptionCode_t & *ex*) const

Returns the ID of a sparse type code member identified by the given index.

The method can be invoked on sparse **DDS_TypeCode** (p. 992) objects.

This function is an RTI Connext extension to the CORBA Type Code Specification.

Precondition:

self kind is **DDS_TK_SPARSE** (p. 67).
Member index in the interval [0,(member count-1)].

MT Safety:

SAFE.

Parameters:

index *<<**in**>>* *(p. 200)* Member index in the interval [0,(member count-1)].

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BOUNDS_USER_EXCEPTION_CODE** (p. 313)

Returns:

ID of the member if no errors.

6.153.2.29 DDS_UnsignedLong DDS_TypeCode::find_member_by_id (DDS_Long *id*, DDS_ExceptionCode_t & *ex*) const

Get the index of the member of the given ID.

MT Safety:

SAFE.

6.153.2.30 DDS_UnsignedLong DDS_TypeCode::add_member_to_enum (const char * *name*, DDS_Long *ordinal*, DDS_ExceptionCode_t & *ex*)

Add a new enumerated constant to this enum **DDS_TypeCode** (p. 992).

This method is applicable to **DDS_TypeCode** (p. 992) objects representing enumerations (**DDS_TK_ENUM** (p. 66)). To add a field to a structured type, see **DDS_TypeCode::add_member_to_enum** (p. 1016).

Modifying a **DDS_TypeCode** (p. 992) – such as by adding a member – is important if you are using the **Dynamic Data** (p. 77) APIs.

MT Safety:

UNSAFE.

Parameters:

name *<<**in**>> (p. 200)* The name of the new member. This string must be unique within this type and must not be NULL.

ordinal *<<**in**>> (p. 200)* The relative order of the new member in this enum or a custom integer value. The value must be unique within the type.

ex *<<**out**>> (p. 200)* If this method fails, this argument will contain information about the failure. Possible values include:

- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_MEMBER_NAME_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_MEMBER_ID_USER_EXCEPTION_CODE** (p. 313)

Returns:

The zero-based index of the new member relative to any other members that previously existed.

See also:

DDS_TypeCode::add_member (p. 1017)
DDS_TypeCode::add_member_ex (p. 1019)
DDS_TypeCodeFactory (p. 1022)

6.153.2.31 DDS_UnsignedLong DDS_TypeCode::add_member (const char * *name*, DDS_Long *id*, const DDS_TypeCode * *tc*, DDS_Octet *member_flags*, DDS_ExceptionCode_t & *ex*)

Add a new member to this **DDS_TypeCode** (p. 992).

This method is applicable to **DDS_TypeCode** (p. 992) objects representing structures (**DDS_TK_STRUCT** (p. 66)), value types (**DDS_TK_VALUE**

(p. 67)), sparse value types (**DDS_TK_SPARSE** (p. 67)), and unions (**DDS_TK_UNION** (p. 66)). To add a constant to an enumeration, see **DDS_TypeCode::add_member_to_enum** (p. 1016).

Modifying a **DDS_TypeCode** (p. 992) – such as by adding a member – is important if you are using the **Dynamic Data** (p. 77) APIs.

Here's a simple code example that adds two fields to a data type, one an integer and another a sequence of integers.

```
// Integer:
 myTypeCode->add_member(
     "myFieldName",
     // If the type is sparse, specify an ID. Otherwise, use this sentinel:
     DDS_TYPECODE_MEMBER_ID_INVALID,
     DDS_TheTypeCodeFactory->get_primitive_tc(DDS_TK_LONG),
     // New field is not a key:
     DDS_TYPECODE_NONKEY_REQUIRED_MEMBER);

 // Sequence of 10 or fewer integers:
 myTypeCode.add_member(
     "myFieldName",
     // If the type is sparse, specify an ID. Otherwise, use this sentinel:
     DDS_TYPECODE_MEMBER_ID_INVALID,
     DDS_TheTypeCodeFactory->create_sequence_tc(
         10,
         DDS_TheTypeCodeFactory->get_primitive_tc(DDS_TK_LONG)),
     // New field is not a key:
     DDS_TYPECODE_NONKEY_REQUIRED_MEMBER);
```

MT Safety:

UNSAFE.

Parameters:

name *<<**in**>>* (p. 200) The name of the new member.

id *<<**in**>>* (p. 200) The ID of the new member. This should only be specified for members of kind **DDS_TK_SPARSE** (p. 67) and **DDS_TK_UNION** (p. 66); otherwise, it should be **DDS_TYPECODE_MEMBER_ID_INVALID** (p. 62).

tc *<<**in**>>* (p. 200) The type of the new member. You can get or create this **DDS_TypeCode** (p. 992) with the **DDS_TypeCodeFactory** (p. 1022).

member_flags *<<**in**>>* (p. 200) Indicates whether the member is part of the key and whether it is required.

ex *<<**out**>>* (p. 200) If this method fails, this argument will contain information about the failure. Possible values include:

- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)

- ^ **DDS_BAD_MEMBER_NAME_USER_EXCEPTION_CODE** (p. 313)
- ^ **DDS_BAD_MEMBER_ID_USER_EXCEPTION_CODE** (p. 313)

Returns:

The zero-based index of the new member relative to any other members that previously existed.

See also:

DDS_TypeCode::add_member_ex (p. 1019)
DDS_TypeCode::add_member_to_enum (p. 1016)
DDS_TypeCodeFactory (p. 1022)
DDS_TYPECODE_NONKEY_MEMBER (p. 64)
DDS_TYPECODE_KEY_MEMBER (p. 64)
DDS_TYPECODE_NONKEY_REQUIRED_MEMBER (p. 65)

6.153.2.32 DDS_UnsignedLong DDS_TypeCode::add_member_ex (const char * *name*, DDS_Long *id*, const DDS_TypeCode * *tc*, DDS_Octet *member_flags*, DDS_Visibility *visibility*, DDS_Boolean *is_pointer*, DDS_Short *bits*, DDS_ExceptionCode_t & *ex*)

Add a new member to this **DDS_TypeCode** (p. 992).

Modifying a **DDS_TypeCode** (p. 992) – such as by adding a member – is important if you are using the **Dynamic Data** (p. 77) APIs.

MT Safety:

UNSAFE.

Parameters:

name *<<in>> (p. 200)* The name of the new member.

id *<<in>> (p. 200)* The ID of the new member. This should only be specified for members of kind **DDS_TK_SPARSE** (p. 67) and **DDS_TK_UNION** (p. 66); otherwise, it should be **DDS_TYPECODE_MEMBER_ID_INVALID** (p. 62).

tc *<<in>> (p. 200)* The type of the new member. You can get or create this **DDS_TypeCode** (p. 992) with the **DDS_TypeCodeFactory** (p. 1022).

member_flags *<<in>> (p. 200)* Indicates whether the member is part of the key and whether it is required.

visibility *<<in>> (p. 200)* Whether the new member is public or private. Non-public members are only relevant for types of kind **DDS_TK_VALUE** (p. 67) and **DDS_TK_SPARSE** (p. 67). Possible values include:

- **DDS_PRIVATE_MEMBER** (p. 63)
- **DDS_PUBLIC_MEMBER** (p. 63)

is_pointer *<<in>> (p. 200)* Whether the data member, in its deserialized form, should be stored by pointer as opposed to by value.

bits *<<in>> (p. 200)* The number of bits, if this new member is a bit field, or DDS_TypeCode::NOT_BITFIELD.

ex *<<out>> (p. 200)* If this method fails, this argument will contain information about the failure. Possible values include:

- **DDS_BADKIND_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_MEMBER_NAME_USER_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_MEMBER_ID_USER_EXCEPTION_CODE** (p. 313)

Returns:

The zero-based index of the new member relative to any other members that previously existed.

See also:

DDS_TypeCode::add_member (p. 1017)
DDS_TypeCodeFactory (p. 1022)
DDS_TYPECODE_NONKEY_MEMBER (p. 64)
DDS_TYPECODE_KEY_MEMBER (p. 64)
DDS_TYPECODE_NONKEY_REQUIRED_MEMBER (p. 65)

6.153.2.33 void DDS_TypeCode::print_IDL (DDS_UnsignedLong *indent*, DDS_ExceptionCode_t & *ex*) const

Prints a **DDS_TypeCode** (p. 992) in a pseudo-IDL notation.

MT Safety:

SAFE.

Parameters:

indent *<<in>> (p. 200)* Indent.

ex <<***out***>> *(*p. *200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

6.154 DDS_TypeCodeFactory Struct Reference

A singleton factory for creating, copying, and deleting data type definitions dynamically.

Public Member Functions

- **DDS_TypeCode** * **clone_tc** (const **DDS_TypeCode** *tc, **DDS_ExceptionCode_t** &ex)

  Creates and returns a copy of the input ***DDS_TypeCode*** *(p. 992).*

- void **delete_tc** (**DDS_TypeCode** *tc, **DDS_ExceptionCode_t** &ex)

  Deletes the input ***DDS_TypeCode*** *(p. 992).*

- const **DDS_TypeCode** * **get_primitive_tc** (**DDS_TCKind** tc_kind)

  Get the ***DDS_TypeCode*** *(p. 992) for a primitive type (integers, floating point values, etc.) identified by the given* ***DDS_TCKind*** *(p. 66).*

- **DDS_TypeCode** * **create_struct_tc** (const char *name, const **DDS_StructMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  Constructs a ***DDS_TK_STRUCT*** *(p. 66)* ***DDS_TypeCode*** *(p. 992).*

- **DDS_TypeCode** * **create_struct_tc** (const char *name, **DDS_ExtensibilityKind** extensibility_kind, const **DDS_StructMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  Constructs a ***DDS_TK_STRUCT*** *(p. 66)* ***DDS_TypeCode*** *(p. 992).*

- **DDS_TypeCode** * **create_value_tc** (const char *name, **DDS_ValueModifier** type_modifier, const **DDS_TypeCode** *concrete_base, const **DDS_ValueMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  Constructs a ***DDS_TK_VALUE*** *(p. 67)* ***DDS_TypeCode*** *(p. 992).*

- **DDS_TypeCode** * **create_value_tc** (const char *name, **DDS_ExtensibilityKind** extensibility_kind, **DDS_ValueModifier** type_modifier, const **DDS_TypeCode** *concrete_base, const **DDS_ValueMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  Constructs a ***DDS_TK_VALUE*** *(p. 67)* ***DDS_TypeCode*** *(p. 992).*

- **DDS_TypeCode** * **create_union_tc** (const char *name, const **DDS_TypeCode** *discriminator_type, **DDS_Long** default_index, const **DDS_UnionMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  Constructs a ***DDS_TK_UNION*** *(p. 66)* ***DDS_TypeCode*** *(p. 992).*

- **DDS_TypeCode** * **create_union_tc** (const char *name, **DDS_ExtensibilityKind** extensibility_kind, const **DDS_TypeCode** *discriminator_type, **DDS_Long** default_index, const **DDS_UnionMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_UNION*** (p. 66) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_enum_tc** (const char *name, const **DDS_EnumMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_ENUM*** (p. 66) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_enum_tc** (const char *name, **DDS_ExtensibilityKind** extensibility_kind, const **DDS_EnumMemberSeq** &members, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_ENUM*** (p. 66) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_alias_tc** (const char *name, const **DDS_TypeCode** *original_type, **DDS_Boolean** is_pointer, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_ALIAS*** (p. 66) *(typedef)* ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_string_tc** (**DDS_UnsignedLong** bound, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_STRING*** (p. 66) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_wstring_tc** (**DDS_UnsignedLong** bound, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_WSTRING*** (p. 67) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_sequence_tc** (**DDS_UnsignedLong** bound, const **DDS_TypeCode** *element_type, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_SEQUENCE*** (p. 66) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_array_tc** (const **DDS_UnsignedLongSeq** &dimensions, const **DDS_TypeCode** *element_type, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_ARRAY*** (p. 66) ***DDS_TypeCode*** (p. 992).

- **DDS_TypeCode** * **create_array_tc** (**DDS_UnsignedLong** length, const **DDS_TypeCode** *element_type, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_ARRAY*** (p. 66) ***DDS_TypeCode*** (p. 992) *for a single-dimensional array.*

- **DDS_TypeCode** * **create_sparse_tc** (const char *name, **DDS_ValueModifier** type_modifier, const **DDS_TypeCode** *concrete_base, **DDS_ExceptionCode_t** &ex)

  *Constructs a **DDS_TK_SPARSE** (p. 67) **DDS_TypeCode** (p. 992).*

Static Public Member Functions

- static **DDS_TypeCodeFactory** * **get_instance** ()

  Gets the singleton instance of this class.

6.154.1 Detailed Description

A singleton factory for creating, copying, and deleting data type definitions dynamically.

You can access the singleton with the **DDS_TypeCodeFactory::get_instance** (p. 1025) method.

If you want to publish and subscribe to data of types that are not known to you at system design time, this class will be your starting point. After creating a data type definition with this class, you will modify that definition using the **DDS_TypeCode** (p. 992) class and then register it with the **Dynamic Data** (p. 77) API.

The methods of this class fall into several categories:

Getting definitions for primitive types:

Type definitions for primitive types (e.g. integers, floating point values, etc.) are pre-defined; your application only needs to *get* them, not *create* them.

- **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026)

Creating definitions for strings, arrays, and sequences:

Type definitions for strings, arrays, and sequences (i.e. variables-size lists) must be created as you need them, because the type definition includes the maximum length of those containers.

- **DDS_TypeCodeFactory::create_string_tc** (p. 1033)
- **DDS_TypeCodeFactory::create_wstring_tc** (p. 1034)
- **DDS_TypeCodeFactory::create_array_tc** (p. 1035)

- **DDS_TypeCodeFactory::create_array_tc** (p. 1035)
- **DDS_TypeCodeFactory::create_sequence_tc** (p. 1034)

Creating definitions for structured types:

Structured types include structures, value types, sparse value types, and unions.

- **DDS_TypeCodeFactory::create_struct_tc** (p. 1027)
- **DDS_TypeCodeFactory::create_value_tc** (p. 1028)
- **DDS_TypeCodeFactory::create_sparse_tc** (p. 1036)
- **DDS_TypeCodeFactory::create_union_tc** (p. 1030)

Creating definitions for other types:

The type system also supports enumerations and aliases (i.e. `typedefs` in C and C++).

- **DDS_TypeCodeFactory::create_enum_tc** (p. 1032)
- **DDS_TypeCodeFactory::create_alias_tc** (p. 1033)

Deleting type definitions:

When you're finished using a type definition, you should delete it. (*Note* that you only need to delete a **DDS_TypeCode** (p. 992) that you *created*; if you got the object from **DDS_TypeCodeFactory::get_primitive_tc** (p. 1026), you must *not* delete it.)

- **DDS_TypeCodeFactory::delete_tc** (p. 1026)

Copying type definitions:

You can also create deep copies of type definitions:

- **DDS_TypeCodeFactory::clone_tc** (p. 1026)

6.154.2 Member Function Documentation

6.154.2.1 static DDS_TypeCodeFactory* DDS_TypeCodeFactory::get_instance () [static]

Gets the singleton instance of this class.

Returns:

The **DDS_TypeCodeFactory** (p. 1022) instance if no errors. Otherwise, NULL.

6.154.2.2 DDS_TypeCode* DDS_TypeCodeFactory::clone_tc (const DDS_TypeCode * *tc*, DDS_ExceptionCode_t & *ex*)

Creates and returns a copy of the input **DDS_TypeCode** (p. 992).

Parameters:

tc <<***in***>> *(p. 200)* Type code that will be copied. Cannot be NULL.

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A clone of tc.

6.154.2.3 void DDS_TypeCodeFactory::delete_tc (DDS_TypeCode * *tc*, DDS_ExceptionCode_t & *ex*)

Deletes the input **DDS_TypeCode** (p. 992).

All the type codes created through the **DDS_TypeCodeFactory** (p. 1022) must be deleted using this method.

Parameters:

tc <<***inout***>> *(p. 200)* Type code that will be deleted. Cannot be NULL.

ex <<***out***>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)

6.154.2.4 const DDS_TypeCode* DDS_TypeCodeFactory::get_primitive_tc (DDS_TCKind *tc_kind*)

Get the **DDS_TypeCode** (p. 992) for a primitive type (integers, floating point values, etc.) identified by the given **DDS_TCKind** (p. 66).

This method is equivalent to, and replaces, the DDS_g_tc_* constants.

See also:

DDS_g_tc_long (p. 68)
DDS_g_tc_ulong (p. 69)
DDS_g_tc_short
DDS_g_tc_ushort (p. 68)
DDS_g_tc_float (p. 69)
DDS_g_tc_double (p. 69)
DDS_g_tc_longdouble (p. 70)
DDS_g_tc_octet (p. 70)
DDS_g_tc_boolean (p. 69)
DDS_g_tc_char
DDS_g_tc_wchar (p. 71)

6.154.2.5 DDS_TypeCode* DDS_TypeCodeFactory::create_struct_tc (const char * *name*, const DDS_StructMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_STRUCT** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

name <<*in*>> *(p. 200)* Name of the struct type. Cannot be NULL.

members <<*in*>> *(p. 200)* Initial members of the structure. This list may be empty (that is, **FooSeq::length()** (p. 1501) may return zero). If the list is not empty, the elements must describe valid struct members. (For example, the names must be unique within the type.)

ex <<*out*>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a struct.

6.154.2.6 DDS_TypeCode* DDS_TypeCodeFactory::create_struct_tc (const char * *name*, DDS_ExtensibilityKind *extensibility_kind*, const DDS_StructMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_STRUCT** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>> (p. 200)* Name of the struct type. Cannot be NULL.

extensibility_kind *<<in>> (p. 200)* Type extensibility.

members *<<in>> (p. 200)* Initial members of the structure. This list may be empty (that is, **FooSeq::length()** (p. 1501) may return zero). If the list is not empty, the elements must describe valid struct members. (For example, the names must be unique within the type.)

ex *<<out>> (p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a struct.

6.154.2.7 DDS_TypeCode* DDS_TypeCodeFactory::create_value_tc (const char * *name*, DDS_ValueModifier *type_modifier*, const DDS_TypeCode * *concrete_base*, const DDS_ValueMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_VALUE** (p. 67) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>> (p. 200)* Name of the value type. Cannot be NULL.

type_modifier *<<in>> (p. 200)* One of the value type modifier constants: **DDS_VM_NONE** (p. 62), **DDS_VM_CUSTOM** (p. 62), **DDS_VM_ABSTRACT** (p. 63) or **DDS_VM_TRUNCATABLE** (p. 63).

concrete_base *<<in>> (p. 200)* **DDS_TypeCode** (p. 992) object describing the concrete valuetype base. It may be NULL if the valuetype does not have a concrete base.

members *<<in>> (p. 200)* Initial members of the value type. This list may be empty. If the list is not empty, the elements must describe valid value type members. (For example, the names must be unique within the type.)

ex *<<out>> (p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a value.

6.154.2.8 DDS_TypeCode* DDS_TypeCodeFactory::create_value_tc (const char * *name*, DDS_ExtensibilityKind *extensibility_kind*, DDS_ValueModifier *type_modifier*, const DDS_TypeCode * *concrete_base*, const DDS_ValueMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_VALUE** (p. 67) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>> (p. 200)* Name of the value type. Cannot be NULL.

extensibility_kind *<<in>> (p. 200)* Type extensibility.

type_modifier *<<in>> (p. 200)* One of the value type modifier constants: **DDS_VM_NONE** (p. 62), **DDS_VM_CUSTOM** (p. 62), **DDS_VM_ABSTRACT** (p. 63) or **DDS_VM_TRUNCATABLE** (p. 63).

concrete_base *<<in>> (p. 200)* **DDS_TypeCode** (p. 992) object describing the concrete valuetype base. It may be NULL if the valuetype does not have a concrete base.

members *<<**in**>>* *(p. 200)* Initial members of the value type. This list may be empty. If the list is not empty, the elements must describe valid value type members. (For example, the names must be unique within the type.)

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a value.

6.154.2.9 DDS_TypeCode* DDS_TypeCodeFactory::create_union_tc (const char * *name*, const DDS_TypeCode * *discriminator_type*, DDS_Long *default_index*, const DDS_UnionMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_UNION** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

name *<<**in**>>* *(p. 200)* Name of the union type. Cannot be NULL.

discriminator_type *<<**in**>>* *(p. 200)* Discriminator Type Code. Cannot be NULL.

default_index *<<**in**>>* *(p. 200)* Index of the default member, or -1 if there is no default member.

members *<<**in**>>* *(p. 200)* Initial members of the union. This list may be empty. If the list is not empty, the elements must describe valid struct members. (For example, the names must be unique within the type.)

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)

- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a union.

6.154.2.10 DDS_TypeCode* DDS_TypeCodeFactory::create_union_tc (const char * *name*, DDS_ExtensibilityKind *extensibility_kind*, const DDS_TypeCode * *discriminator_type*, DDS_Long *default_index*, const DDS_UnionMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_UNION** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>>* (p. 200) Name of the union type. Cannot be NULL.

extensibility_kind *<<in>>* (p. 200) Type extensibility.

discriminator_type *<<in>>* (p. 200) Discriminator Type Code. Cannot be NULL.

default_index *<<in>>* (p. 200) Index of the default member, or -1 if there is no default member.

members *<<in>>* (p. 200) Initial members of the union. This list may be empty. If the list is not empty, the elements must describe valid struct members. (For example, the names must be unique within the type.)

ex *<<out>>* (p. 200) Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a union.

6.154.2.11 DDS_TypeCode* DDS_TypeCodeFactory::create_enum_tc (const char * *name*, const DDS_EnumMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_ENUM** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>>* *(p. 200)* Name of the enum type. Cannot be NULL.

members *<<in>>* *(p. 200)* Initial members of the enumeration. All members must have non-NULL names, and both names and ordinal values must be unique within the type. Note that it is also possible to add members later with **DDS_TypeCode::add_member_to_enum** (p. 1016).

ex *<<out>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing an enumeration.

6.154.2.12 DDS_TypeCode* DDS_TypeCodeFactory::create_enum_tc (const char * *name*, DDS_ExtensibilityKind *extensibility_kind*, const DDS_EnumMemberSeq & *members*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_ENUM** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>>* *(p. 200)* Name of the enum type. Cannot be NULL.

members *<<in>>* *(p. 200)* Initial members of the enumeration. All members must have non-NULL names, and both names and ordinal values must be unique within the type. Note that it is also possible to add members later with **DDS_TypeCode::add_member_to_enum** (p. 1016).

ex *<<out>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing an enumeration.

6.154.2.13 DDS_TypeCode* DDS_TypeCodeFactory::create_alias_tc (const char * *name*, const DDS_TypeCode * *original_type*, DDS_Boolean *is_pointer*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_ALIAS** (p. 66) (typedef) **DDS_TypeCode** (p. 992).

Parameters:

name *<<in>> (p. 200)* Name of the alias. Cannot be NULL.

original_type *<<in>> (p. 200)* Aliased type code. Cannot be NULL.

is_pointer *<<in>> (p. 200)* Indicates if the alias is a pointer to the aliased type code.

ex *<<out>> (p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing an alias.

6.154.2.14 DDS_TypeCode* DDS_TypeCodeFactory::create_string_tc (DDS_UnsignedLong *bound*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_STRING** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

bound *<<**in**>>* *(p. 200)* Maximum length of the string.

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a string.

6.154.2.15 DDS_TypeCode* DDS_TypeCodeFactory::create_wstring_tc (DDS_UnsignedLong *bound*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_WSTRING** (p. 67) **DDS_TypeCode** (p. 992).

Parameters:

bound *<<**in**>>* *(p. 200)* Maximum length of the wide string.

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a wide string.

6.154.2.16 DDS_TypeCode* DDS_TypeCodeFactory::create_sequence_tc (DDS_UnsignedLong *bound*, const DDS_TypeCode * *element_type*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_SEQUENCE** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

bound <<*in*>> *(p. 200)* The bound for the sequence (> 0).

element_type <<*in*>> *(p. 200)* **DDS_TypeCode** (p. 992) object describing the sequence elements.

ex <<*out*>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a sequence.

6.154.2.17 DDS_TypeCode* DDS_TypeCodeFactory::create_array_tc (const DDS_UnsignedLongSeq & *dimensions*, const DDS_TypeCode * *element_type*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_ARRAY** (p. 66) **DDS_TypeCode** (p. 992).

Parameters:

dimensions <<*in*>> *(p. 200)* Dimensions of the array. Each dimension has to be greater than 0.

element_type <<*in*>> *(p. 200)* **DDS_TypeCode** (p. 992) describing the array elements. Cannot be NULL.

ex <<*out*>> *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a sequence.

6.154.2.18 DDS_TypeCode* DDS_TypeCodeFactory::create_array_tc (DDS_UnsignedLong *length*, const DDS_TypeCode * *element_type*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_ARRAY** (p. 66) **DDS_TypeCode** (p. 992) for a single-dimensional array.

Parameters:

length *<<in>> (p. 200)* Length of the single-dimensional array.

element_type *<<in>> (p. 200)* **DDS_TypeCode** (p. 992) describing the array elements. Cannot be NULL.

ex *<<out>> (p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a sequence.

6.154.2.19 DDS_TypeCode* DDS_TypeCodeFactory::create_sparse_tc (const char * *name*, DDS_ValueModifier *type_modifier*, const DDS_TypeCode * *concrete_base*, DDS_ExceptionCode_t & *ex*)

Constructs a **DDS_TK_SPARSE** (p. 67) **DDS_TypeCode** (p. 992).

A sparse value type is similar to other value types but with one major difference: not all members need to be present in every sample.

It is not possible to generate code for sparse value types; they must be created at runtime using these APIs. You will interact with samples of sparse types using the **Dynamic Data** (p. 77) APIs.

Parameters:

name *<<in>> (p. 200)* Name of the value type. Cannot be NULL.

type_modifier *<<in>> (p. 200)* One of the value type modifier constants: **DDS_VM_NONE** (p. 62), **DDS_VM_CUSTOM** (p. 62),

DDS_VM_ABSTRACT (p. 63) or **DDS_VM_TRUNCATABLE** (p. 63).

concrete_base *<<**in**>>* *(p. 200)* **DDS_TypeCode** (p. 992) object describing the concrete valuetype base. It may be NULL if the valuetype does not have a concrete base.

ex *<<**out**>>* *(p. 200)* Parameter for error indications. The values that can take are:

- **DDS_NO_EXCEPTION_CODE** (p. 312)
- **DDS_BAD_PARAM_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_NO_MEMORY_SYSTEM_EXCEPTION_CODE** (p. 313)
- **DDS_BAD_TYPECODE_SYSTEM_EXCEPTION_CODE** (p. 313)

Returns:

A newly-created **DDS_TypeCode** (p. 992) object describing a value.

6.155 DDS_TypeConsistencyEnforcementQosPolicy Struct Reference

Defines the rules for determining whether the type used to publish a given topic is consistent with that used to subscribe to it.

Public Attributes

- **DDS_TypeConsistencyKind kind**
  Type consistency kind.

6.155.1 Detailed Description

Defines the rules for determining whether the type used to publish a given topic is consistent with that used to subscribe to it.

This policy defines a type consistency kind, which allows applications to select from among a set of predetermined policies. The following consistency kinds are specified: **DDS_DISALLOW_TYPE_COERCION** (p. 433) and **DDS_ALLOW_TYPE_COERCION** (p. 433).

The default enforcement kind is **DDS_ALLOW_TYPE_COERCION** (p. 433). However, when the middleware is introspecting the built-in topic data declaration of a remote DataReader in order to determine whether it can match with a local DataWriter, if it observes that no TypeConsistencyEnforcementQosPolicy value is provided (as would be the case when communicating with a Service implementation not in conformance with this specification), it assumes a kind of **DDS_DISALLOW_TYPE_COERCION** (p. 433).

Entity:

DDSDataReader (p. 1087)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **UNTIL ENABLE** (p. 340)

6.155.2 Member Data Documentation

6.155.2.1 DDS_TypeConsistencyKind DDS_TypeConsistencyEnforcementQosPolicy::kind

Type consistency kind.

[default] **DDS_ALLOW_TYPE_COERCION** (p. 433)

6.156 DDS_TypeSupportQosPolicy Struct Reference

Allows you to attach application-specific values to a DataWriter or DataReader that are passed to the serialization or deserialization routine of the associated data type.

Public Attributes

- void * **plugin_data**

  Value to pass into the type plugin's de-/serialization function.

6.156.1 Detailed Description

Allows you to attach application-specific values to a DataWriter or DataReader that are passed to the serialization or deserialization routine of the associated data type.

The purpose of this QoS is to allow a user application to pass data to a type plugin's support functions.

Entity:

DDSDataReader (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = N/A

Changeable (p. 340) = **YES** (p. 340)

6.156.2 Usage

This QoS policy allows you to associate a pointer to an object with a **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087). This object pointer is passed to the serialization routine of the data type associated with the **DDSDataWriter** (p. 1113) or the deserialization routine of the data type associated with the **DDSDataReader** (p. 1087).

You can modify the rtiddsgen-generated code so that the de/serialization routines act differently depending on the information passed in via the object pointer. (The generated serialization and deserialization code does not use the pointer.)

This functionality can be used to change how data sent by a **DDSDataWriter** (p. 1113) or received by a **DDSDataReader** (p. 1087) is serialized or deserialized on a per DataWriter and DataReader basis.

It can also be used to dynamically change how serialization (or for a less common case, deserialization) occurs. For example, a data type could represent a table, including the names of the rows and columns. However, since the row/column names of an instance of the table (a Topic) don't change, they only need to be sent once. The information passed in through the TypeSupport QoS policy could be used to signal the serialization routine to send the row/column names the first time a **DDSDataWriter** (p. 1113) calls **FooDataWriter::write** (p. 1484), and then never again.

6.156.3 Member Data Documentation

6.156.3.1 void* DDS_TypeSupportQosPolicy::plugin_data

Value to pass into the type plugin's de-/serialization function.

[default] NULL

6.157 DDS_UnionMember Struct Reference

A description of a member of a union.

Public Attributes

- char * **name**

  The name of the union member.

- **DDS_Boolean is_pointer**

  Indicates whether the union member is a pointer or not.

- struct **DDS_LongSeq labels**

  The labels of the union member.

- const **DDS_TypeCode** * **type**

  The type of the union member.

6.157.1 Detailed Description

A description of a member of a union.

See also:

DDS_UnionMemberSeq (p. 1044)
DDS_TypeCodeFactory::create_union_tc (p. 1030)

6.157.2 Member Data Documentation

6.157.2.1 char* DDS_UnionMember::name

The name of the union member.

Cannot be NULL.

6.157.2.2 DDS_Boolean DDS_UnionMember::is_pointer

Indicates whether the union member is a pointer or not.

6.157.2.3 struct DDS_LongSeq DDS_UnionMember::labels [read]

The labels of the union member.

Each union member should contain at least one label. If the union discriminator type is not **DDS_Long** (p. 300) the label value should be evaluated to an integer value. For instance, 'a' would be evaluated to 97.

6.157.2.4 const DDS_TypeCode* DDS_UnionMember::type

The type of the union member.

Cannot be NULL.

6.158 DDS_UnionMemberSeq Struct Reference

Defines a sequence of union members.

6.158.1 Detailed Description

Defines a sequence of union members.

See also:

DDS_UnionMember (p. 1042)
FooSeq (p. 1494)
DDS_TypeCodeFactory::create_union_tc (p. 1030)

6.159 DDS_UnsignedLongLongSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_UnsignedLongLong** (p. 300) >.

6.159.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_UnsignedLongLong** (p. 300) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_UnsignedLongLong (p. 300)
FooSeq (p. 1494)

6.160 DDS_UnsignedLongSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_UnsignedLong** (p. 300) >.

6.160.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_UnsignedLong** (p. 300) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_UnsignedLong (p. 300)
FooSeq (p. 1494)

6.161 DDS_UnsignedShortSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDS_UnsignedShort** (p. 299) >.

6.161.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDS_UnsignedShort** (p. 299) >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_UnsignedShort (p. 299)
FooSeq (p. 1494)

6.162 DDS_UserDataQosPolicy Struct Reference

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Public Attributes

- struct **DDS_OctetSeq value**
  a sequence of octets

6.162.1 Detailed Description

Attaches a buffer of opaque data that is distributed by means of **Built-in Topics** (p. 42) during discovery.

Entity:

DDSDomainParticipant (p. 1139), **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113)

Properties:

RxO (p. 340) = NO;
Changeable (p. 340) = **YES** (p. 340)

See also:

DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.162.2 Usage

The purpose of this QoS is to allow the application to attach additional information to the created **DDSEntity** (p. 1253) objects, so that when a remote application discovers their existence, it can access that information and use it for its own purposes. This information is not used by RTI Connext.

One possible use of this QoS is to attach security credentials or some other information that can be used by the remote application to authenticate the source.

In combination with operations such as **DDSDomainParticipant::ignore_participant** (p. 1187), **DDSDomainParticipant::ignore_publication**

(p. 1189), **DDSDomainParticipant::ignore_subscription** (p. 1190), and **DDSDomainParticipant::ignore_topic** (p. 1188), this QoS policy can assist an application to define and enforce its own security policies.

The use of this QoS is not limited to security; it offers a simple, yet flexible extensibility mechanism.

Important: RTI Connext stores the data placed in this policy in pre-allocated pools. It is therefore necessary to configure RTI Connext with the maximum size of the data that will be stored in policies of this type. This size is configured with **DDS_DomainParticipantResourceLimitsQosPolicy::participant_user_data_max_length** (p. 606), **DDS_DomainParticipantResourceLimitsQosPolicy::writer_user_data_max_length** (p. 607), and **DDS_DomainParticipantResourceLimitsQosPolicy::reader_user_data_max_length** (p. 607).

6.162.3 Member Data Documentation

6.162.3.1 struct DDS_OctetSeq DDS_UserDataQosPolicy::value [read]

a sequence of octets

[**default**] empty (zero-length)

[**range**] Octet sequence of length [0,max_length]

6.163 DDS_ValueMember Struct Reference

A description of a member of a value type.

Public Attributes

- char * **name**

  The name of the value member.

- const **DDS_TypeCode** * **type**

  The type of the value member.

- **DDS_Boolean is_pointer**

  Indicates whether the value member is a pointer or not.

- **DDS_Short bits**

  Number of bits of a bitfield member.

- **DDS_Boolean is_key**

  Indicates if the value member is a key member or not.

- **DDS_Visibility access**

  The type of access (public, private) for the value member.

6.163.1 Detailed Description

A description of a member of a value type.

See also:

DDS_ValueMemberSeq (p. 1052)
DDS_TypeCodeFactory::create_value_tc (p. 1028)

6.163.2 Member Data Documentation

6.163.2.1 char* DDS_ValueMember::name

The name of the value member.

Cannot be NULL.

6.163.2.2 const DDS_TypeCode* DDS_ValueMember::type

The type of the value member.

Cannot be NULL.

6.163.2.3 DDS_Boolean DDS_ValueMember::is_pointer

Indicates whether the value member is a pointer or not.

6.163.2.4 DDS_Short DDS_ValueMember::bits

Number of bits of a bitfield member.

If the struct member is a bitfield, this field contains the number of bits of the bitfield. Otherwise, bits should contain DDS_TypeCode::NOT_BITFIELD.

6.163.2.5 DDS_Boolean DDS_ValueMember::is_key

Indicates if the value member is a key member or not.

6.163.2.6 DDS_Visibility DDS_ValueMember::access

The type of access (public, private) for the value member.

It can take the values: **DDS_PRIVATE_MEMBER** (p. 63) or **DDS_PUBLIC_MEMBER** (p. 63).

6.164 DDS_ValueMemberSeq Struct Reference

Defines a sequence of value members.

6.164.1 Detailed Description

Defines a sequence of value members.

See also:

DDS_ValueMember (p. 1050)
FooSeq (p. 1494)
DDS_TypeCodeFactory::create_value_tc (p. 1028)

6.165 DDS_VendorId_t Struct Reference

<<*eXtension*>> *(p. 199)* Type used to represent the vendor of the service implementing the RTPS protocol.

Public Attributes

- **DDS_Octet vendorId** [DDS_VENDOR_ID_LENGTH_MAX]

  The vendor Id.

6.165.1 Detailed Description

<<*eXtension*>> *(p. 199)* Type used to represent the vendor of the service implementing the RTPS protocol.

6.165.2 Member Data Documentation

6.165.2.1 DDS_Octet DDS_VendorId_t::vendorId[DDS_VENDOR_ID_LENGTH_MAX]

The vendor Id.

6.166 DDS_VirtualSubscriptionBuiltinTopicData Struct Reference

Public Attributes

- **DDS_BuiltinTopicKey_t key**
- char * **topic_name**
- char * **name**
- **DDS_Long quorum**

6.166.1 Detailed Description

6.166.2 Member Data Documentation

6.166.2.1 DDS_BuiltinTopicKey_t DDS_VirtualSubscriptionBuiltinTopicData::key

6.166.2.2 char* DDS_VirtualSubscriptionBuiltinTopicData::topic_name

6.166.2.3 char* DDS_VirtualSubscriptionBuiltinTopicData::name

6.166.2.4 DDS_Long DDS_VirtualSubscriptionBuiltinTopicData::quorum

6.167 DDS_VirtualSubscriptionBuiltinTopicDataSeq Struct Reference

6.167.1 Detailed Description

6.168 DDS_WaitSetProperty_t Struct Reference

<<eXtension>> (p. 199) Specifies the **DDSWaitSet** (p. 1433) behavior for multiple trigger events.

Public Attributes

- long **max_event_count**

  *Maximum number of trigger events to cause a **DDSWaitSet** (p. 1433) to awaken.*

- struct **DDS_Duration_t max_event_delay**

  *Maximum delay from occurrence of first trigger event to cause a **DDSWaitSet** (p. 1433) to awaken.*

6.168.1 Detailed Description

<<eXtension>> (p. 199) Specifies the **DDSWaitSet** (p. 1433) behavior for multiple trigger events.

In simple use, a **DDSWaitSet** (p. 1433) returns when a single trigger event occurs on one of its attached **DDSCondition** (p. 1075) (s), or when the `timeout` maximum wait duration specified in the **DDSWaitSet::wait** (p. 1438) call expires.

The **DDS_WaitSetProperty_t** (p. 1056) allows configuration of the waiting behavior of a **DDSWaitSet** (p. 1433). If no conditions are true at the time of the call to wait, then the `max_event_count` parameter may be used to configure the WaitSet to wait for `max_event_count` trigger events to occur before returning, or to wait for up to `max_event_delay` time from the occurrence of the first trigger event before returning.

The `timeout` maximum wait duration specified in the **DDSWaitSet::wait** (p. 1438) call continues to apply.

Entity:

DDSWaitSet (p. 1433)

Properties:

RxO (p. 340) = N/A

Changeable (p. 340) = **YES** (p. 340)

6.168.2 Member Data Documentation

6.168.2.1 long DDS_WaitSetProperty_t::max_event_count

Maximum number of trigger events to cause a **DDSWaitSet** (p. 1433) to awaken.

The **DDSWaitSet** (p. 1433) will wait until up to `max_event_count` trigger events have occurred before returning. The **DDSWaitSet** (p. 1433) may return earlier if either the `timeout` duration has expired, or `max_event_delay` has elapsed since the occurrence of the first trigger event. `max_event_count` may be used to "collect" multiple trigger events for processing at the same time.

[**default**] 1

[**range**] >= 1

;

6.168.2.2 struct DDS_Duration_t DDS_WaitSetProperty_t::max_event_delay [read]

Maximum delay from occurrence of first trigger event to cause a **DDSWaitSet** (p. 1433) to awaken.

The **DDSWaitSet** (p. 1433) will return no later than `max_event_delay` after the first trigger event. `max_event_delay` may be used to establish a maximum latency for events reported by the **DDSWaitSet** (p. 1433).

Note that **DDS_RETCODE_TIMEOUT** (p. 316) is *not* returned if `max_event_delay` is exceeded. **DDS_RETCODE_TIMEOUT** (p. 316) is returned only if the `timeout` duration expires before any trigger events occur.

[**default**] **DDS_DURATION_INFINITE** (p. 305);

6.169 DDS_WcharSeq Struct Reference

Instantiates FooSeq (p. 1494) < **DDS_Wchar** (p. 299) >.

6.169.1 Detailed Description

Instantiates FooSeq (p. 1494) < **DDS_Wchar** (p. 299) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Wchar (p. 299)
FooSeq (p. 1494)

6.170 DDS_WireProtocolQosPolicy Struct Reference

Specifies the wire-protocol-related attributes for the **DDSDomainParticipant** (p. 1139).

Public Attributes

- **DDS_Long participant_id**
  A value used to distinguish among different participants belonging to the same domain on the same host.
- **DDS_UnsignedLong rtps_host_id**
  The RTPS Host ID of the domain participant.
- **DDS_UnsignedLong rtps_app_id**
  The RTPS App ID of the domain participant.
- **DDS_UnsignedLong rtps_instance_id**
  *The RTPS Instance ID of the **DDSDomainParticipant** (p. 1139).*
- struct **DDS_RtpsWellKnownPorts_t rtps_well_known_ports**
  Configures the RTPS well-known port mappings.
- **DDS_RtpsReservedPortKindMask rtps_reserved_port_mask**
  Specifies which well-known ports to reserve when enabling the participant.
- **DDS_WireProtocolQosPolicyAutoKind rtps_auto_id_kind**
  Kind of auto mechanism used to calculate the GUID prefix.

6.170.1 Detailed Description

Specifies the wire-protocol-related attributes for the **DDSDomainParticipant** (p. 1139).

Entity:

DDSDomainParticipant (p. 1139)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **NO** (p. 340)

6.170.2 Usage

This QoS policy configures some participant-wide properties of the DDS Real-Time Publish Subscribe (RTPS) on-the-wire protocol. (**DDS_DataWriterProtocolQosPolicy** (p. 535) and **DDS_DataReaderProtocolQosPolicy** (p. 501) configure RTPS and reliability properties on a per **DDSDataWriter** (p. 1113) or **DDSDataReader** (p. 1087) basis.)

NOTE: The default QoS policies returned by RTI Connext contain the correctly initialized wire protocol attributes. The defaults are not normally expected to be modified, but are available to the advanced user customizing the implementation behavior.

The default values should not be modified without an understanding of the underlying Real-Time Publish Subscribe (RTPS) wire protocol.

In order for the discovery process to work correctly, each **DDSDomainParticipant** (p. 1139) must have a unique identifier. This QoS policy specifies how that identifier should be generated.

RTPS defines a 96-bit prefix to this identifier; each **DDSDomainParticipant** (p. 1139) must have a unique value for this prefix relative to all other participants in its domain. To make it easier to control how this 96-bit value is generated, RTI Connext divides it into three integers: a *host ID*, the value of which is based on the identity of the identity of the machine on which the participant is executing; an *application ID*, the value of which is based on the process or task in which the particpant is contained; and an *instance ID*, which identifies the participant itself.

This QoS policy provides you with a choice of algorithms for generating these values automatically. In case none of these algorithms suit your needs, you may also choose to specify some or all of them yourself.

The following three fields:

- **DDS_WireProtocolQosPolicy::rtps_host_id** (p. 1064)
- **DDS_WireProtocolQosPolicy::rtps_app_id** (p. 1064)
- **DDS_WireProtocolQosPolicy::rtps_instance_id** (p. 1065)

comprise the GUID prefix and by default are set to **DDS_RTPS_AUTO_ID** (p. **??**). The meaning of this flag depends on the value assigned to the **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) field.

Depending on the **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) value, there are two different scenarios:

1. In the default and most common scenario, **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is set to

DDS_RTPS_AUTO_ID_FROM_IP (p. 404). Doing so, each field is interpreted as follows:

- **rtps_host_id:** The 32-bit value of the IPv4 of the first up and running interface of the host machine is assigned.
- **rtps_app_id:** The process (or task) ID is assigned.
- **rtps_instance_id:** A counter is assigned that is incremented per new participant.

NOTE: If the IP assigned to the interface is not unique within the network (for instance, if it is not configured), it is possible that the GUID (specifically, the rtps_host_id portion) may also not be unique.

2. In this situation, RTI Connext provides a different value for rtps_auto_id_kind: **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404). As the name suggests, this alternative mechanism uses the MAC address instead of the IPv4 address. Since the MAC address size is up to 64 bits, the logical mapping of the host information, the application ID, and the instance identifiers has to change.

   Note to Solaris Users: To use **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404), you must run the RTI Connext application while logged in as root.

   Using **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404), the default value of each field is interpreted as follows:

   - **rtps_host_id:** The first 32 bits of the MAC address of the first up and running interface of the host machine are assigned.
   - **rtps_app_id:** The last 32 bits of the MAC address of the first up and running interface of the host machine are assigned.
   - **rtps_instance_id:** This field is split into two different parts. The process (or task) ID is assigned to the first 24 bits. A counter is assigned to the last 8 bits. This counter is incremented per new participant. In both scenarios, you can change the value of each field independently.

   If **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404) is used, the rtps_instance_id has been logically split into two parts: 24 bits for the process/task ID and 8 bits for the per new participant counter. To give to users the ability to manually set the two parts independently, a bitfield mechanism has been introduced for the rtps_instance_id field when it is used in combination with **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404). If one of the two parts is set to 0, only this part will be handled by RTI Connext and you will be able to handle the other one manually.

Some examples are provided to clarify the behavior of this QoSPolicy in case you want to change the default behavior with **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404).

First, get the participant QoS from the DomainParticipantFactory:

```
DDSDomainParticipantFactory *factory = NULL;
 factory = DDSTheParticipantFactory->get_instance();
 factory->get_default_participant_qos(participant_qos);
```

Second, change the **DDS_WireProtocolQosPolicy** (p. 1059) using one of the options shown below.

Third, create the **DDSDomainParticipant** (p. 1139) as usual, using the modified QoS structure instead of the default one.

Option 1: Use **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404) to explicitly set just the application/task identifier portion of the **rtps_instance_id** field.

```
participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
 participant_qos.wire_protocol.rtps_host_id      = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_app_id       = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_instance_id  = (/* App ID */ (12 << 8) |
                                                   /* Instance ID*/ (DDS_RTPS_AUTO_ID));
```

Option 2: Handle only the per participant counter and let RTI Connext handle the application/task identifier:

```
participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
 participant_qos.wire_protocol.rtps_host_id      = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_app_id       = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_instance_id  =  (/* App ID */ (DDS_RTPS_AUTO_ID) |
                                                   /* Instance ID*/ (12));
```

Option 3: Handle the entire **rtps_instance_id** field yourself:

```
participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
 participant_qos.wire_protocol.rtps_host_id      = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_app_id       = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_instance_id  = ( /* App ID */ (12 << 8)) |
                                                    /* Instance ID */ (9) )
```

NOTE: If you are using **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404) as **rtps_auto_id_kind** and you decide to manually handle the **rtps_instance_id** field, you must ensure that both parts are non-zero (otherwise RTI Connext will take responsibility for them). RTI recommends that you always specify the two parts separately in order to avoid errors.

Option 4: Let RTI Connext handle the entire **rtps_instance_id** field:

```
participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
 participant_qos.wire_protocol.rtps_host_id      = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_app_id       = DDS_RTPS_AUTO_ID;
 participant_qos.wire_protocol.rtps_instance_id  = DDS_RTPS_AUTO_ID;
```

NOTE: If you are using **DDS_RTPS_AUTO_ID_FROM_MAC** (p. 404) as **rtps_auto_id_kind** and you decide to manually handle the **rtps_instance_id** field, you must ensure that both parts are non-zero (otherwise RTI Connext will take responsibility for them). RTI recommends that you always specify the two parts separately in order to clearly show the difference.

6.170.3 Member Data Documentation

6.170.3.1 DDS_Long DDS_WireProtocolQosPolicy::participant_id

A value used to distinguish among different participants belonging to the same domain on the same host.

Determines the unicast port on which meta-traffic is received. Also defines the *default* unicast port for receiving user-traffic for DataReaders and DataWriters (can be overridden by the **DDS_DataReaderQos::unicast** (p. 519) or **DDS_DataWriterQos::unicast** (p. 558)).

For more information on port mapping, please refer to **DDS_RtpsWellKnownPorts_t** (p. 905).

Each **DDSDomainParticipant** (p. 1139) in the same domain and running on the same host, must have a unique `participant_id`. The participants may be in the same address space or in distinct address spaces.

A negative number (-1) means that RTI Connext will *automatically* resolve the participant ID as follows.

- RTI Connext will pick the *smallest* participant ID based on the unicast ports available on the transports enabled for discovery.
- RTI Connext will attempt to resolve an automatic port index either when a DomainPartcipant is enabled, or when a DataReader or DataWriter is created. Therefore, all the transports enabled for discovery must have been registered by this time. Otherwise, the discovery transports registered after resolving the automatic port index may produce port conflicts when the DomainPartcipant is enabled.

[**default**] -1 [automatic], i.e. RTI Connext will automatically pick the `participant_id`, as described above.

[**range**] [>= 0], or -1, and does not violate guidelines stated in **DDS_RtpsWellKnownPorts_t** (p. 905).

See also:

DDSEntity::enable() (p. 1256)
NDDSTransportSupport::register_transport() (p. 1560)

6.170.3.2 DDS_UnsignedLong DDS_WireProtocolQosPolicy::rtps_host_id

The RTPS Host ID of the domain participant.

A machine/operating system specific host ID that is unique in the domain.

[default] DDS_RTPS_AUTO_ID (p. **??**). The default value is interpreted as follows:

If **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is equal to *RTPS_AUTO_ID_FROM_IP* (the default value), the value will be interpreted as the IPv4 address of the *first* up and running interface of the host machine.

If **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is equal to *RTPS_AUTO_ID_FROM_MAC*, the value will be interpreted as the first 32 bits of the MAC address assigned to the *first* up and running interface of the host machine.

[range] [0,0xffffffff]

6.170.3.3 DDS_UnsignedLong DDS_WireProtocolQosPolicy::rtps_app_id

The RTPS App ID of the domain participant.

A participant specific ID that, together with the rtps_instance_id, is unique within the scope of the rtps_host_id.

If a participant dies and is restarted, it is recommended that it be given an app ID that is distinct from the previous one so that other participants in the domain can distinguish between them.

[default] DDS_RTPS_AUTO_ID (p. **??**). The default value is interpreted as follows:

If **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is equal to *RTPS_AUTO_ID_FROM_IP* (default value for this field) the value will be the process (or task) ID.

If **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is equal to *RTPS_AUTO_ID_FROM_MAC* the value will be the last 32 bits of the MAC address assigned to the *first* up and running interface of the host machine.
[range] [0,0xffffffff]

6.170.3.4 DDS_UnsignedLong DDS_WireProtocolQosPolicy::rtps_instance_id

The RTPS Instance ID of the **DDSDomainParticipant** (p. 1139).

This is an instance-specific ID of a participant that, together with the rtps_app_id, is unique within the scope of the rtps_host_id.

If a participant dies and is restarted, it is recommended that it be given an instance ID that is distinct from the previous one so that other participants in the domain can distinguish between them.

[default] DDS_RTPS_AUTO_ID (p. **??**). The default value is interpreted as follows:

If **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is equal to *RTPS_AUTO_ID_FROM_IP* (the default value), a counter is assigned that is incremented per new participant. For VxWorks-653, the first 8 bits are assigned to the partition id for the application.

If **DDS_WireProtocolQosPolicy::rtps_auto_id_kind** (p. 1066) is equal to *RTPS_AUTO_ID_FROM_MAC*, the first 24 bits are assigned to the application/task identifier and the last 8 bits are assigned to a counter that is incremented per new paricipant.

[range] [0,0xffffffff] **NOTE:** If you use *DDS_RTPS_AUTO_ID_FROM_MAC* as **rtps_auto_id_kind** and you decide to manually handle the **rtps_instance_id** field, you must ensure that both the two parts are non-zero, otherwise the middleware will take responsibility for them. We recommend that you always specify the two parts separately in order to avoid errors. (`examples`)

6.170.3.5 struct DDS_RtpsWellKnownPorts_t DDS_WireProtocolQosPolicy::rtps_well_known_ports [read]

Configures the RTPS well-known port mappings.

Determines the well-known multicast and unicast port mappings for discovery (meta) traffic and user traffic.

[default] DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS (p. 405)

6.170.3.6 DDS_RtpsReservedPortKindMask DDS_WireProtocolQosPolicy::rtps_reserved_port_mask

Specifies which well-known ports to reserve when enabling the participant.

Specifies which of the well-known multicast and unicast ports will be reserved

when the DomainParticipant is enabled. Failure to allocate a port that is computed based on the **DDS_RtpsWellKnownPorts_t** (p. 905) will be detected at this time, and the enable operation will fail.

[default] **DDS_RTPS_RESERVED_PORT_MASK_DEFAULT** (p. 401)

6.170.3.7 DDS_WireProtocolQosPolicyAutoKind DDS_WireProtocolQosPolicy::rtps_auto_id_kind

Kind of auto mechanism used to calculate the GUID prefix.

[default] RTPS_AUTO_ID_FROM_IP

6.171 DDS_WriteParams_t Struct Reference

<<*eXtension*>> *(p. 199)* Input parameters for writing with **FooDataWriter::write_w_params** (p. 1487), **FooDataWriter::dispose_w_params** (p. 1491), **FooDataWriter::register_instance_w_params** (p. 1480), **FooDataWriter::unregister_instance_w_params** (p. 1483)

Public Attributes

- **DDS_Boolean replace_auto**

  Allows retrieving the actual value of those fields that were automatic.

- struct **DDS_SampleIdentity_t identity**

  Identity of the sample.

- struct **DDS_SampleIdentity_t related_sample_identity**

  The identity of another sample related to this one.

- struct **DDS_Time_t source_timestamp**

  Source timestamp upon write.

- **DDS_InstanceHandle_t handle**

  Instance handle.

- **DDS_Long priority**

  Publication priority.

6.171.1 Detailed Description

<<*eXtension*>> *(p. 199)* Input parameters for writing with **FooDataWriter::write_w_params** (p. 1487), **FooDataWriter::dispose_w_params** (p. 1491), **FooDataWriter::register_instance_w_params** (p. 1480), **FooDataWriter::unregister_instance_w_params** (p. 1483)

6.171.2 Member Data Documentation

6.171.2.1 DDS_Boolean DDS_WriteParams_t::replace_auto

Allows retrieving the actual value of those fields that were automatic.

When this field is set, the fields that were configured with an automatic value (for example, **DDS_AUTO_SAMPLE_IDENTITY** (p. 449)) receive their actual value after **FooDataWriter::write_w_params** (p. 1487) is called.

To reset those fields to their automatic value after calling **FooDataWriter::write_w_params** (p. 1487), use **DDS_WriteParams_reset** (p. 448)

6.171.2.2 struct DDS_SampleIdentity_t DDS_WriteParams_t::identity [read]

Identity of the sample.

Identifies the sample being written. The identity consists of a pair (Virtual Writer GUID, Virtual Sequence Number).

Use the default value to let RTI Connext determine the sample identity as follows:

- The Virtual Writer GUID is the virtual GUID associated with the writer writing the sample. This virtual GUID is configured using **DDS_DataWriterProtocolQosPolicy::virtual_guid** (p. 536).
- The sequence number is increased by one with respect to the previous value.

The virtual sequence numbers for a virtual writer must be strictly monotonically increasing. If the user tries to write a sample with a sequence number smaller or equal to the last sequence number, the write operation will fail.

A DataReader can access the identity of a received sample by using the fields **DDS_SampleInfo::original_publication_virtual_guid** (p. 920) and **DDS_SampleInfo::original_publication_virtual_sequence_number** (p. 920) in the **DDS_SampleInfo** (p. 912).

[default] **DDS_AUTO_SAMPLE_IDENTITY** (p. 449).

6.171.2.3 struct DDS_SampleIdentity_t DDS_WriteParams_t::related_sample_identity [read]

The identity of another sample related to this one.

Identifies another sample that is logically related to the one that is written.

When this field is set, the related sample identity is propagated an subscribing applications can retrieve it from the **DDS_SampleInfo** (p. 912) (see **DDS_SampleInfo_get_related_sample_identity** (p. 110)).

The default value is **DDS_UNKNOWN_SAMPLE_IDENTITY** (p. 449), and is not propagated.

A DataReader can access the related identity of a received sample by using the fields **DDS_SampleInfo::related_original_publication_virtual_guid** (p. 921) and **DDS_SampleInfo::related_original_publication_virtual_sequence_number** (p. 921) in the **DDS_SampleInfo** (p. 912).

[default] DDS_UNKNOWN_SAMPLE_IDENTITY (p. 449)

6.171.2.4 struct DDS_Time_t DDS_WriteParams_t::source_timestamp [read]

Source timestamp upon write.

Specifies the source timestamp that will be available to the **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute within the **DDS_SampleInfo** (p. 912).

[default] DDS_TIME_INVALID (p. 305).

6.171.2.5 DDS_InstanceHandle_t DDS_WriteParams_t::handle

Instance handle.

Either the handle returned by a previous call to **FooDataWriter::register_instance** (p. 1478), or else the special value **DDS_HANDLE_NIL** (p. 55).

[default] DDS_HANDLE_NIL (p. 55)

6.171.2.6 DDS_Long DDS_WriteParams_t::priority

Publication priority.

A positive integer value designating the relative priority of the sample, used to determine the transmission order of pending writes.

Use of publication priorities requires an asynchronous publisher (**DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422)) with **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) set to **DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY** (p. 90).

Larger numbers have higher priority.

For multi-channel DataWriters, the publication priority of a sample may be used as a filter criteria for determining channel membership.

If the publication priority of the parent DataWriter, or for multi-channel DataWriters, if the publication priority of the parent channel, is set

to **DDS_PUBLICATION_PRIORITY_AUTOMATIC** (p. 421), then the DataWriter or channel will be assigned the priority of the largest publication priority of all samples in the DataWriter or channel.

If the publication priority of the parent DataWriter, and for multi-channel DataWriters, if the publication priority of the parent channel, are set to **DDS_PUBLICATION_PRIORITY_UNDEFINED** (p. 421), then the DataWriter or channel will be assigned the lowest priority, regardless of the value of the publication priorities of samples written to the DataWriter or channel.

The publication priority of each sample can be set in the **DDS_WriteParams_t** (p. 1067) of **FooDataWriter::write_w_params** (p. 1487).

For dispose and unregister samples, use the **DDS_WriteParams_t** (p. 1067) of **FooDataWriter::dispose_w_params** (p. 1491) and **FooDataWriter::unregister_instance_w_params** (p. 1483).

[default] 0 (lowest priority)

6.172 DDS_WriterDataLifecycleQosPolicy Struct Reference

Controls how a **DDSDataWriter** (p. 1113) handles the lifecycle of the instances (keys) that it is registered to manage.

Public Attributes

- **DDS_Boolean autodispose_unregistered_instances**

  *Boolean flag that controls the behavior when the **DDSDataWriter** (p. 1113) unregisters an instance by means of the unregister operations.*

- struct **DDS_Duration_t autopurge_unregistered_instances_delay**

  <<**eXtension**>> (p. 199) *Maximum duration for which the **DDSDataWriter** (p. 1113) will maintain information regarding an instance once it has unregistered the instance.*

6.172.1 Detailed Description

Controls how a **DDSDataWriter** (p. 1113) handles the lifecycle of the instances (keys) that it is registered to manage.

Entity:

DDSDataWriter (p. 1113)

Properties:

RxO (p. 340) = N/A
Changeable (p. 340) = **YES** (p. 340)

6.172.2 Usage

This policy determines how the **DDSDataWriter** (p. 1113) acts with regards to the lifecycle of the data instances it manages (data instances that have been either explicitly registered with the **DDSDataWriter** (p. 1113) or implicitly registered by directly writing the data).

Since the deletion of a DataWriter automatically unregisters all data instances it manages, the setting of the `autodispose_unregistered_instances` flag will only determine whether instances are ultimately disposed when the **DDSDataWriter** (p. 1113) is deleted either directly by means of the **DDSPublisher::delete_datawriter** (p. 1358) operation or indirectly as a consequence

of calling **DDSPublisher::delete_contained_entities** (p. 1363) or **DDSDomainParticipant::delete_contained_entities** (p. 1193) that contains the DataWriter.

You may use **FooDataWriter::unregister_instance** (p. 1480) to indicate that the **DDSDataWriter** (p. 1113) no longer wants to send data for a **DDSTopic** (p. 1419).

The behavior controlled by this QoS policy applies on a per instance (key) basis for keyed Topics, so that when a **DDSDataWriter** (p. 1113) unregisters an instance, RTI Connext can automatically also dispose that instance. This is the default behavior.

In many cases where the ownership of a Topic is shared (see **DDS_OwnershipQosPolicy** (p. 807)), DataWriters may want to relinquish their ownership of a particular instance of the Topic to allow other DataWriters to send updates for the value of that instance regardless of Ownership Strength. In that case, you may only want a DataWriter to unregister an instance without disposing the instance. *Disposing* an instance is a statement that an instance no longer exists. User applications may be coded to trigger on the disposal of instances, thus the ability to unregister without disposing may be useful to properly maintain the semantic of disposal.

6.172.3 Member Data Documentation

6.172.3.1 DDS_Boolean DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances

Boolean flag that controls the behavior when the **DDSDataWriter** (p. 1113) unregisters an instance by means of the unregister operations.

- **DDS_BOOLEAN_TRUE** (p. 298) (default)

  The **DDSDataWriter** (p. 1113) will dispose the instance each time it is unregistered. The behavior is identical to explicitly calling one of the `dispose` operations on the instance prior to calling the `unregister` operation.

- **DDS_BOOLEAN_FALSE** (p. 299)

  The **DDSDataWriter** (p. 1113) will not dispose the instance. The application can still call one of the `dispose` operations prior to unregistering the instance and accomplish the same effect.

[default] DDS_BOOLEAN_TRUE (p. 298)

6.172.3.2 struct DDS_Duration_t DDS_WriterDataLifecycleQosPolicy::autopurge_unregistered_instances_delay [read]

<<*eXtension*>> *(p. 199)* Maximum duration for which the **DDSDataWriter** (p. 1113) will maintain information regarding an instance once it has unregistered the instance.

After this time elapses, the **DDSDataWriter** (p. 1113) will purge all internal information regarding the instance, including historical samples.

When the duration is zero, the instance is purged as soon as all the samples have been acknowledged by all the live DataReaders.

[**default**] **DDS_DURATION_INFINITE** (p. 305) (disabled)

[**range**] [0, 1 year] or **DDS_DURATION_INFINITE** (p. 305)

6.173 DDS_WstringSeq Struct Reference

Instantiates FooSeq (p. 1494) < **DDS_Wchar** (p. 299)* >.

6.173.1 Detailed Description

Instantiates FooSeq (p. 1494) < **DDS_Wchar** (p. 299)* >.

Instantiates:

<<*generic*>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDS_Wchar (p. 299)
DDS_StringSeq (p. 929)
FooSeq (p. 1494)

6.174 DDSCondition Class Reference

<<*interface*>> *(p. 199)* Root class for all the conditions that may be attached to a **DDSWaitSet** (p. 1433).

Inheritance diagram for DDSCondition::

Public Member Functions

- virtual **DDS_Boolean get_trigger_value** ()=0

  Retrieve the `trigger_value`.

6.174.1 Detailed Description

<<*interface*>> *(p. 199)* Root class for all the conditions that may be attached to a **DDSWaitSet** (p. 1433).

This basic class is specialised in three classes:

DDSGuardCondition (p. 1263), **DDSStatusCondition** (p. 1376), and **DDSReadCondition** (p. 1374).

A **DDSCondition** (p. 1075) has a `trigger_value` that can be **DDS_BOOLEAN_TRUE** (p. 298) or **DDS_BOOLEAN_FALSE** (p. 299) and is set automatically by RTI Connext.

See also:

DDSWaitSet (p. 1433)

6.174.2 Member Function Documentation

6.174.2.1 virtual DDS_Boolean DDSCondition::get_trigger_value () [pure virtual]

Retrieve the `trigger_value`.

Returns:

the trigger value.

Implemented in **DDSGuardCondition** (p. 1264).

6.175 DDSConditionSeq Struct Reference

Instantiates `FooSeq` (p. 1494) < **DDSCondition** (p. 1075) >.

6.175.1 Detailed Description

Instantiates `FooSeq` (p. 1494) < **DDSCondition** (p. 1075) >.

Instantiates:

<<***generic***>> *(p. 199)* **FooSeq** (p. 1494)

See also:

DDSWaitSet (p. 1433)
FooSeq (p. 1494)

6.176 DDSContentFilter Class Reference

<<*interface*>> *(p. 199)* Interface to be used by a custom filter of a **DDSContentFilteredTopic** (p. 1081)

Inheritance diagram for DDSContentFilter::

Public Member Functions

- virtual **DDS_ReturnCode_t compile** (void **new_compile_data, const char *expression, const **DDS_StringSeq** &parameters, const **DDS_TypeCode** *type_code, const char *type_class_name, void *old_compile_data)=0

  Compile an instance of the content filter according to the filter expression and parameters of the given data type.

- virtual **DDS_Boolean evaluate** (void *compile_data, const void *sample, const struct **DDS_FilterSampleInfo** *meta_data)=0

  Evaluate whether the sample is passing the filter or not according to the sample content.

- virtual void **finalize** (void *compile_data)=0

  A previously compiled instance of the content filter is no longer in use and resources can now be cleaned up.

6.176.1 Detailed Description

<<*interface*>> *(p. 199)* Interface to be used by a custom filter of a **DDSContentFilteredTopic** (p. 1081)

Entity:

DDSContentFilteredTopic (p. 1081)

This is the interface that can be implemented by an application-provided class and then registered with the **DDSDomainParticipant** (p. 1139) such that samples can be filtered for a **DDSContentFilteredTopic** (p. 1081) with the given filter name.

Note: the API for using a custom content filter is subject to change in a future release.

See also:

DDSContentFilteredTopic (p. 1081)
DDSDomainParticipant::register_contentfilter (p. 1156)

6.176.2 Member Function Documentation

6.176.2.1 virtual DDS_ReturnCode_t DDSContentFilter::compile (void ** *new_compile_data*, const char * *expression*, const DDS_StringSeq & *parameters*, const DDS_TypeCode * *type_code*, const char * *type_class_name*, void * *old_compile_data*) [pure virtual]

Compile an instance of the content filter according to the filter expression and parameters of the given data type.

This method is called when an instance of the locally registered content filter is created or when the expression parameter for the locally registered content filter instance is changed.

An instance of the locally registered content filter is created every time a local **DDSContentFilteredTopic** (p. 1081) with the matching filter name is created, or when a **DDSDataReader** (p. 1087) with a matching filter name is discovered.

It is possible for multiple threads to be calling into this function at the same time. However, this function will never be called on a content filter that has been unregistered.

Parameters:

new_compile_data *<<out>>* *(p. 200)* User specified opaque pointer of this instance of the content filter. This value is then passed to the **DDSContentFilter::evaluate** (p. 1079) and **DDSContentFilter::finalize** (p. 1080) functions for this instance of the content filter. Can be set to NULL.

expression *<<in>>* *(p. 200)* An ASCIIZ string with the filter expression. The memory used by this string is owned by RTI Connext and **must** not be freed. If you want to manipulate this string, you **must** first make a copy of it.

parameters *<<in>>* *(p. 200)* A string sequence with the expression parameters the **DDSContentFilteredTopic** (p. 1081) was created with. The string sequence is **equal** (but **not** identical) to the string sequence passed to **DDSDomainParticipant::create_contentfilteredtopic** (p. 1179). Note that the sequence passed to the compile function is **owned** by RTI Connext and **must not** be referenced outside the compile function.

type_code *<<in>>* *(p. 200)* A pointer to the type code for the related **DDSTopic** (p. 1419) of the **DDSContentFilteredTopic** (p. 1081). A type_code is a description of a type in terms of which **types** it contains (such as long, string, etc.) and the corresponding member field names in the data type structure. The type code can be used to write custom content filters that can be used with any type.

type_class_name *<<in>>* *(p. 200)* Fully qualified class name of the related **DDSTopic** (p. 1419).

old_compile_data *<<in>>* *(p. 200)* The previous new_compile_data value from a previous call to this instance of a content filter. If the compile function is called more than once for an instance of a **DDSContentFilteredTopic** (p. 1081), e.g., if the expression parameters are changed, then the new_compile_data value returned by the previous invocation is passed in the old_compile_data parameter (which can be NULL). If this is a new instance of the filter, NULL is passed. This parameter is useful for freeing or reusing resources previously allocated for this

Returns:

One of the **Standard Return Codes** (p. 314)

6.176.2.2 virtual DDS_Boolean DDSContentFilter::evaluate (void * *compile_data*, const void * *sample*, const struct DDS_FilterSampleInfo * *meta_data*) [pure virtual]

Evaluate whether the sample is passing the filter or not according to the sample content.

This method is called when a sample for a locally created **DDSDataReader** (p. 1087) associated with the filter is received, or when a sample for a discovered **DDSDataReader** (p. 1087) assocated with the filter needs to be sent.

It is possible for multiple threads to be calling into this function at the same time. However, this function will never be called on a content filter that has been unregistered.

Parameters:

compile_data *<<in>>* *(p. 200)* The last return value of the **DDSContentFilter::compile** (p. 1078) function for this instance of the content filter. Can be NULL.

sample *<<in>>* *(p. 200)* Pointer to a deserialized sample to be filtered

Returns:

The function must return 0 if the sample should be filtered out, non zero otherwise

6.176.2.3 virtual void DDSContentFilter::finalize (void * *compile_data*) [pure virtual]

A previously compiled instance of the content filter is no longer in use and resources can now be cleaned up.

This method is called when an instance of the locally registered content filter is deleted.

An instance of the locally registered content filter is deleted every time a local **DDSContentFilteredTopic** (p. 1081) with the matching filter name is deleted, or when a **DDSDataReader** (p. 1087) with a matching filter name is removed due to discovery.

This method is also called on all instances of the discovered **DDSDataReader** (p. 1087) with a matching filter name if the filter is unregistered with **DDSDomainParticipant::unregister_contentfilter** (p. 1158)

It is possible for multiple threads to be calling into this function at the same time. However, this function will never be called on a content filter that has been unregistered.

Parameters:

compile_data *<<in>>* *(p. 200)* The last return value of the **DDSContentFilter::compile** (p. 1078) function for this instance of the content filter. Can be NULL.

6.177 DDSContentFilteredTopic Class Reference

<<***interface***>> *(p. 199)* Specialization of **DDSTopicDescription** (p. 1427) that allows for content-based subscriptions.

Inheritance diagram for DDSContentFilteredTopic::

Public Member Functions

- virtual const char * **get_filter_expression** ()=0

  Get the `filter_expression`*.*

- virtual **DDS_ReturnCode_t** **get_expression_parameters** (**DDS_StringSeq** &parameters)=0

  Get the `expression_parameters`*.*

- virtual **DDS_ReturnCode_t** **set_expression_parameters** (const **DDS_StringSeq** &parameters)=0

  Set the `expression_parameters`*.*

- virtual **DDS_ReturnCode_t** **append_to_expression_parameter** (const **DDS_Long** index, const char *val)=0

  <<**eXtension**>> (p. 199) *Appends a string term to the specified parameter string.*

- virtual **DDS_ReturnCode_t** **remove_from_expression_parameter** (const **DDS_Long** index, const char *val)=0

  <<**eXtension**>> (p. 199) *Removes a string term from the specified parameter string.*

- virtual **DDSTopic** * **get_related_topic** ()=0

  Get the `related_topic`*.*

Static Public Member Functions

- static **DDSContentFilteredTopic** * **narrow** (**DDSTopicDescription** *topic_description)

  Narrow the given ***DDSTopicDescription*** *(p. 1427) pointer to a* ***DDSContentFilteredTopic*** *(p. 1081) pointer.*

6.177.1 Detailed Description

<<*interface*>> *(p. 199)* Specialization of **DDSTopicDescription** (p. 1427) that allows for content-based subscriptions.

It describes a more sophisticated subscription that indicates a **DDSDataReader** (p. 1087) does not want to necessarily see all values of each instance published under the **DDSTopic** (p. 1419). Rather, it wants to see only the values whose contents satisfy certain criteria. This class therefore can be used to request content-based subscriptions.

The selection of the content is done using the `filter_expression` with parameters `expression_parameters`.

- The `filter_expression` attribute is a string that specifies the criteria to select the data samples of interest. It is similar to the WHERE part of an SQL clause.

- The `expression_parameters` attribute is a sequence of strings that give values to the 'parameters' (i.e. "%n" tokens) in the `filter_expression`. The number of supplied parameters must fit with the requested values in the `filter_expression` (i.e. the number of n tokens).

Queries and Filters Syntax (p. 208) describes the syntax of `filter_expression` and `expression_parameters`.

Note on Content-Based Filtering and Sparse Value Types

If you are a user of the **Dynamic Data** (p. 77) API, you may define *sparse value types*; that is, types for which every data sample need not include a value for every field defined in the type. (See **DDS_TK_SPARSE** (p. 67) and **DDS_TypeCodeFactory::create_sparse_tc** (p. 1036).) In order for a filter expression on a field to be well defined, that field must be present in the data sample. That means that you will only be able to perform a content-based filter on fields that are marked as **DDS_TYPECODE_KEY_MEMBER** (p. 64) or **DDS_TYPECODE_NONKEY_REQUIRED_MEMBER** (p. 65).

6.177.2 Member Function Documentation

6.177.2.1 static DDSContentFilteredTopic* DDSContentFilteredTopic::narrow (DDSTopicDescription * *topic_description*) [static]

Narrow the given **DDSTopicDescription** (p. 1427) pointer to a **DDSContentFilteredTopic** (p. 1081) pointer.

Returns:

DDSContentFilteredTopic (p. 1081) if this **DDSTopicDescription** (p. 1427) is a **DDSContentFilteredTopic** (p. 1081). Otherwise, return NULL.

6.177.2.2 virtual const char* DDSContentFilteredTopic::get_filter_expression () [pure virtual]

Get the `filter_expression`.

Return the `filter_expression` associated with the **DDSContentFilteredTopic** (p. 1081).

Returns:

the `filter_expression`.

6.177.2.3 virtual DDS_ReturnCode_t DDSContentFilteredTopic::get_expression_parameters (DDS_StringSeq & *parameters*) [pure virtual]

Get the `expression_parameters`.

Return the `expression_parameters` associated with the **DDSContentFilteredTopic** (p. 1081). `expression_parameters` is either specified on the last successful call to **DDSContentFilteredTopic::set_expression_parameters** (p. 1084) or, if that method is never called, the parameters specified when the **DDSContentFilteredTopic** (p. 1081) was created.

Parameters:

parameters *<<inout>> (p. 200)* the filter expression parameters. The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDomainParticipant::create_contentfilteredtopic (p. 1179)
DDSContentFilteredTopic::set_expression_parameters (p. 1084)

6.177.2.4 virtual DDS_ReturnCode_t DDSContentFilteredTopic::set_expression_parameters (const DDS_StringSeq & *parameters*) [pure virtual]

Set the `expression_parameters`.

Change the `expression_parameters` associated with the **DDSContentFilteredTopic** (p. 1081).

Parameters:

parameters *<<in>>* *(p. 200)* the filter expression parameters . Length of sequence cannot be greater than 100.

Returns:

One of the **Standard Return Codes** (p. 314)

6.177.2.5 virtual DDS_ReturnCode_t DDSContentFilteredTopic::append_to_expression_parameter (const DDS_Long *index*, const char * *val*) [pure virtual]

*<<**eXtension**>>* *(p. 199)* Appends a string term to the specified parameter string.

Appends the input string to the end of the specified parameter string, separated by a comma. If the original parameter string is enclosed in quotation marks ("), the resultant string will also be enclosed in quotation marks.

This method can be used in expression parameters associated with MATCH operators in order to add a pattern to the match pattern list. For example, if the filter expression parameter value is:

'IBM'

Then append_to_expression_parameter(0, "MSFT") would generate the new value:

'IBM,MSFT'

Parameters:

index *<<in>>* *(p. 200)* The index of the parameter string to be modified. The first index is index 0. When using the **DDS_STRINGMATCHFILTER_NAME** (p. 41) filter, `index` *must* be 0.

val *<<in>>* *(p. 200)* The string term to be appended to the parameter string.

Returns:

One of the **Standard Return Codes** (p. 314)

6.177.2.6 virtual DDS_ReturnCode_t DDSContentFilteredTopic::remove_from_expression_parameter (const DDS_Long *index*, const char * *val*) [pure virtual]

<<eXtension>> *(p. 199)* Removes a string term from the specified parameter string.

Removes the input string from the specified parameter string. To be found and removed, the input string must exist as a complete term, bounded by comma separators or the strong boundary. If the original parameter string is enclosed in quotation marks ("), the resultant string will also be enclosed in quotation marks. If the removed term was the last entry in the string, the result will be a string of empty quotation marks.

This method can be used in expression parameters associated with MATCH operators in order to remove a pattern from the match pattern list. For example, if the filter expression paremeter value is:

'IBM,MSFT'

Then remove_from_expression_parameter(0, "IBM") would generate the expression:

'MSFT'

Parameters:

index *<<in>>* *(p. 200)* The index of the parameter string to be modified. The first index is index 0. When using the **DDS_STRINGMATCHFILTER_NAME** (p. 41) filter, `index` *must* be 0.

val *<<in>>* *(p. 200)* The string term to be removed from the parameter string.

Returns:

One of the **Standard Return Codes** (p. 314)

6.177.2.7 virtual DDSTopic* DDSContentFilteredTopic::get_related_topic () [pure virtual]

Get the `related_topic`.

Return the **DDSTopic** (p. 1419) specified when the **DDSContentFilteredTopic** (p. 1081) was created.

Returns:

The **DDSTopic** (p. 1419) assocated with the **DDSContentFiltered-Topic** (p. 1081).

6.178 DDSDataReader Class Reference

<<***interface***>> *(*p. *199)* Allows the application to: (1) declare the data it wishes to receive (i.e. make a subscription) and (2) access the data received by the attached **DDSSubscriber** (p. 1390).

Inheritance diagram for DDSDataReader::

Public Member Functions

- virtual **DDSReadCondition** * **create_readcondition** (**DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)

  Creates a ***DDSReadCondition*** *(*p. *1374).*

- virtual **DDSQueryCondition** * **create_querycondition** (**DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states, const char *query_expression, const struct **DDS_StringSeq** &query_parameters)

  Creates a ***DDSQueryCondition*** *(*p. *1372).*

- virtual **DDS_ReturnCode_t delete_readcondition** (**DDSReadCondition** *condition)

  Deletes a ***DDSReadCondition*** *(*p. *1374) or* ***DDSQueryCondition*** *(*p. *1372) attached to the* ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t delete_contained_entities** ()

  Deletes all the entities that were created by means of the "create" operations on the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t wait_for_historical_data** (const **DDS_Duration_t** &max_wait)

  Waits until all "historical" data is received for ***DDSDataReader*** *(*p. *1087) entities that have a non-VOLATILE Durability Qos kind.*

- virtual **DDS_ReturnCode_t acknowledge_sample** (const **DDS_SampleInfo** &sample_info)

  Acknowledge a single sample explicitly.

- virtual **DDS_ReturnCode_t acknowledge_all** ()

  Acknowledge all previously accessed samples.

- virtual **DDS_ReturnCode_t acknowledge_sample** (const **DDS_SampleInfo** &sample_info, const **DDS_AckResponseData_t** &response_data)

  *[**Not supported.**] Acknowledge a single sample explicitly*

- virtual **DDS_ReturnCode_t acknowledge_all** (const **DDS_AckResponseData_t** &response_data)

  *[**Not supported.**] Acknowledge all previously accessed samples*

- virtual **DDS_ReturnCode_t get_matched_publications** (**DDS_InstanceHandleSeq** &publication_handles)

  *Retrieve the list of publications currently "associated" with this **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t get_matched_publication_data** (**DDS_PublicationBuiltinTopicData** &publication_data, const **DDS_InstanceHandle_t** &publication_handle)

  *This operation retrieves the information on a publication that is currently "associated" with the **DDSDataReader** (p. 1087).*

- virtual **DDSTopicDescription** * **get_topicdescription** ()

  *Returns the **DDSTopicDescription** (p. 1427) associated with the **DDSDataReader** (p. 1087).*

- virtual **DDSSubscriber** * **get_subscriber** ()

  *Returns the **DDSSubscriber** (p. 1390) to which the **DDSDataReader** (p. 1087) belongs.*

- virtual **DDS_ReturnCode_t get_sample_rejected_status** (**DDS_SampleRejectedStatus** &status)

  *Accesses the **DDS_SAMPLE_REJECTED_STATUS** (p. 324) communication status.*

- virtual **DDS_ReturnCode_t get_liveliness_changed_status** (**DDS_LivelinessChangedStatus** &status)

  *Accesses the **DDS_LIVELINESS_CHANGED_STATUS** (p. 325) communication status.*

- virtual **DDS_ReturnCode_t get_requested_deadline_missed_status** (**DDS_RequestedDeadlineMissedStatus** &status)

  *Accesses the **DDS_REQUESTED_DEADLINE_MISSED_STATUS** (p. 323) communication status.*

- virtual **DDS_ReturnCode_t get_requested_incompatible_qos_status** (**DDS_RequestedIncompatibleQosStatus** &status)

*Accesses the **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323) communication status.*

- virtual **DDS_ReturnCode_t get_sample_lost_status** (**DDS_SampleLostStatus** &status)

  *Accesses the **DDS_SAMPLE_LOST_STATUS** (p. 324) communication status.*

- virtual **DDS_ReturnCode_t get_subscription_matched_status** (**DDS_SubscriptionMatchedStatus** &status)

  *Accesses the **DDS_SUBSCRIPTION_MATCHED_STATUS** (p. 326) communication status.*

- virtual **DDS_ReturnCode_t get_datareader_cache_status** (**DDS_DataReaderCacheStatus** &status)

  <<**eXtension**>> (p. 199) *Get the datareader cache status for this reader.*

- virtual **DDS_ReturnCode_t get_datareader_protocol_status** (**DDS_DataReaderProtocolStatus** &status)

  <<**eXtension**>> (p. 199) *Get the datareader protocol status for this reader.*

- virtual **DDS_ReturnCode_t get_matched_publication_datareader_protocol_status** (**DDS_DataReaderProtocolStatus** &status, const **DDS_InstanceHandle_t** &publication_handle)

  <<**eXtension**>> (p. 199) *Get the datareader protocol status for this reader, per matched publication identified by the publication_handle.*

- virtual **DDS_ReturnCode_t set_qos** (const **DDS_DataReaderQos** &qos)

  Sets the reader QoS.

- virtual **DDS_ReturnCode_t get_qos** (**DDS_DataReaderQos** &qos)

  Gets the reader QoS.

- virtual **DDS_ReturnCode_t set_qos_with_profile** (const char *library_name, const char *profile_name)

  <<**eXtension**>> (p. 199) *Change the QoS of this reader using the input XML QoS profile.*

- virtual **DDS_ReturnCode_t set_listener** (**DDSDataReaderListener** *l, **DDS_StatusMask** mask=DDS_STATUS_MASK_ALL)

  Sets the reader listener.

- virtual **DDSDataReaderListener** * **get_listener** ()

Get the reader listener.

ˆ virtual **DDS_ReturnCode_t enable** ()

*Enables the **DDSEntity** (p. 1253).*

ˆ virtual **DDSStatusCondition** * **get_statuscondition** ()

*Allows access to the **DDSStatusCondition** (p. 1376) associated with the **DDSEntity** (p. 1253).*

ˆ virtual **DDS_StatusMask get_status_changes** ()

*Retrieves the list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.*

ˆ virtual **DDS_InstanceHandle_t get_instance_handle** ()

*Allows access to the **DDS_InstanceHandle_t** (p. 53) associated with the **DDSEntity** (p. 1253).*

6.178.1 Detailed Description

<<***interface***>> *(p. 199)* Allows the application to: (1) declare the data it wishes to receive (i.e. make a subscription) and (2) access the data received by the attached **DDSSubscriber** (p. 1390).

QoS:

DDS_DataReaderQos (p. 515)

Status:

DDS_DATA_AVAILABLE_STATUS (p. 324);
DDS_LIVELINESS_CHANGED_STATUS (p. 325), **DDS_LivelinessChangedStatus** (p. 775);
DDS_REQUESTED_DEADLINE_MISSED_STATUS (p. 323), **DDS_RequestedDeadlineMissedStatus** (p. 875);
DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_RequestedIncompatibleQosStatus** (p. 877);
DDS_SAMPLE_LOST_STATUS (p. 324), **DDS_SampleLostStatus** (p. 923);
DDS_SAMPLE_REJECTED_STATUS (p. 324), **DDS_SampleRejectedStatus** (p. 925);
DDS_SUBSCRIPTION_MATCHED_STATUS (p. 326), DDS_SubscriptionMatchedStatus

Listener:

DDSDataReaderListener (p. 1108)

A **DDSDataReader** (p. 1087) refers to exactly one **DDSTopicDescription** (p. 1427) (either a **DDSTopic** (p. 1419), a **DDSContentFilteredTopic** (p. 1081) or a **DDSMultiTopic** (p. 1322)) that identifies the data to be read.

The subscription has a unique resulting type. The data-reader may give access to several instances of the resulting type, which can be distinguished from each other by their `key`.

DDSDataReader (p. 1087) is an abstract class. It must be specialised for each particular application data-type (see **USER_DATA** (p. 345)). The additional methods or functions that must be defined in the auto-generated class for a hypothetical application type `Foo` (p. 1443) are specified in the generic type **FooDataReader** (p. 1444).

The following operations may be called even if the **DDSDataReader** (p. 1087) is not enabled. Other operations will fail with the value **DDS_RETCODE_NOT_ENABLED** (p. 315) if called on a disabled **DDSDataReader** (p. 1087):

- The base-class operations **DDSDataReader::set_qos** (p. 1102), **DDSDataReader::get_qos** (p. 1103), **DDSDataReader::set_listener** (p. 1104), **DDSDataReader::get_listener** (p. 1104), **DDSEntity::enable** (p. 1256), **DDSEntity::get_statuscondition** (p. 1257) and **DDSEntity::get_status_changes** (p. 1257)

- **DDSDataReader::get_liveliness_changed_status** (p. 1099) **DDSDataReader::get_requested_deadline_missed_status** (p. 1099) **DDSDataReader::get_requested_incompatible_qos_status** (p. 1100) **DDSDataReader::get_sample_lost_status** (p. 1100) **DDSDataReader::get_sample_rejected_status** (p. 1099) **DDSDataReader::get_subscription_matched_status** (p. 1100)

All sample-accessing operations, namely: **FooDataReader::read** (p. 1447), **FooDataReader::take** (p. 1448), **FooDataReader::read_w_condition** (p. 1454), and **FooDataReader::take_w_condition** (p. 1456) may fail with the error **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) as described in **DDSSubscriber::begin_access** (p. 1405).

See also:

Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_subscriber.cxx, and **HelloWorldSupport.cxx**.

6.178.2 Member Function Documentation

6.178.2.1 virtual DDSReadCondition* DDSDataReader::create_readcondition (DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [inline, virtual]

Creates a **DDSReadCondition** (p. 1374).

The returned **DDSReadCondition** (p. 1374) will be attached and belong to the **DDSDataReader** (p. 1087).

Parameters:

sample_states *<<in>> (p. 200)* sample state of the data samples that are of interest

view_states *<<in>> (p. 200)* view state of the data samples that are of interest

instance_states *<<in>> (p. 200)* instance state of the data samples that are of interest

Returns:

return **DDSReadCondition** (p. 1374) created. Returns NULL in case of failure.

6.178.2.2 virtual DDSQueryCondition* DDSDataReader::create_querycondition (DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*, const char * *query_expression*, const struct DDS_StringSeq & *query_parameters*) [inline, virtual]

Creates a **DDSQueryCondition** (p. 1372).

The returned **DDSQueryCondition** (p. 1372) will be attached and belong to the **DDSDataReader** (p. 1087).

Queries and Filters Syntax (p. 208) describes the syntax of `query_expression` and `query_parameters`.

Parameters:

sample_states *<<in>> (p. 200)* sample state of the data samples that are of interest

view_states *<<in>>* *(p. 200)* view state of the data samples that are of interest

instance_states *<<in>>* *(p. 200)* instance state of the data samples that are of interest

query_expression *<<in>>* *(p. 200)* Expression for the query. Cannot be NULL.

query_parameters *<<in>>* *(p. 200)* Parameters for the query expression.

Returns:

return **DDSQueryCondition** (p. 1372) created. Returns NULL in case of failure.

6.178.2.3 virtual DDS_ReturnCode_t DDSDataReader::delete_readcondition (DDSReadCondition * *condition*) [inline, virtual]

Deletes a **DDSReadCondition** (p. 1374) or **DDSQueryCondition** (p. 1372) attached to the **DDSDataReader** (p. 1087).

Since **DDSQueryCondition** (p. 1372) specializes **DDSReadCondition** (p. 1374), it can also be used to delete a **DDSQueryCondition** (p. 1372).

Precondition:

The **DDSReadCondition** (p. 1374) must be attached to the **DDSDataReader** (p. 1087), or the operation will fail with the error **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Parameters:

condition *<<in>>* *(p. 200)* Condition to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

6.178.2.4 virtual DDS_ReturnCode_t DDSDataReader::delete_contained_entities () [inline, virtual]

Deletes all the entities that were created by means of the "create" operations on the **DDSDataReader** (p. 1087).

Deletes all contained **DDSReadCondition** (p. 1374) and **DDSQueryCondition** (p. 1372) objects.

The operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) if the any of the contained entities is in a state where it cannot be deleted.

Once **DDSDataReader::delete_contained_entities** (p. 1093) completes successfully, the application may delete the **DDSDataReader** (p. 1087), knowing that it has no contained **DDSReadCondition** (p. 1374) and **DDSQueryCondition** (p. 1372) objects.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

6.178.2.5 virtual DDS_ReturnCode_t DDSDataReader::wait_for_historical_data (const DDS_Duration_t & *max_wait*) [inline, virtual]

Waits until all "historical" data is received for **DDSDataReader** (p. 1087) entities that have a non-VOLATILE Durability Qos kind.

This operation is intended only for **DDSDataReader** (p. 1087) entities that have a non-VOLATILE Durability QoS kind.

As soon as an application enables a non-VOLATILE **DDSDataReader** (p. 1087), it will start receiving both "historical" data (i.e., the data that was written prior to the time the **DDSDataReader** (p. 1087) joined the domain) as well as any new data written by the **DDSDataWriter** (p. 1113) entities. There are situations where the application logic may require the application to wait until all "historical" data is received. This is the purpose of the **DDSDataReader::wait_for_historical_data** (p. 1094) operations.

The operation **DDSDataReader::wait_for_historical_data** (p. 1094) blocks the calling thread until either all "historical" data is received, or else duration specified by the max_wait parameter clapses, whichever happens first. A successful completion indicates that all the "historical" data was "received"; timing out indicates that max_wait elapsed before all the data was received.

Parameters:

max_wait *<<in>> (p. 200)* Timeout value.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.178.2.6 virtual DDS_ReturnCode_t DDSDataReader::acknowledge_sample (const DDS_SampleInfo & *sample_info*) [inline, virtual]

Acknowledge a single sample explicitly.

Applicable only when **DDS_ReliabilityQosPolicy::acknowledgment_kind** (p. 868) = **DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE** (p. 364)

Parameters:

sample_info <<*in*>> *(p. 200)* **DDS_SampleInfo** (p. 912) identifying the sample being acknowledged.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.7 virtual DDS_ReturnCode_t DDSDataReader::acknowledge_all () [inline, virtual]

Acknowledge all previously accessed samples.

Applicable only when **DDS_ReliabilityQosPolicy::acknowledgment_kind** (p. 868) = **DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE** (p. 364)

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.8 virtual DDS_ReturnCode_t DDSDataReader::acknowledge_sample (const DDS_SampleInfo & *sample_info*, const DDS_AckResponseData_t & *response_data*) [inline, virtual]

[**Not supported.**] Acknowledge a single sample explicitly

Applicable only when **DDS_ReliabilityQosPolicy::acknowledgment_kind** (p. 868) = **DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE** (p. 364)

Parameters:

sample_info *<<in>>* *(p. 200)* **DDS_SampleInfo** (p. 912) identifying the sample being acknowledged.

response_data *<<in>>* *(p. 200)* Response data sent to **DDSDataWriter** (p. 1113) upon acknowledgment

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.9 virtual DDS_ReturnCode_t DDSDataReader::acknowledge_all (const DDS_AckResponseData_t & *response_data*) [inline, virtual]

[**Not supported.**] Acknowledge all previously accessed samples

Applicable only when **DDS_ReliabilityQosPolicy::acknowledgment_kind** (p. 868) = **DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE** (p. 364)

Parameters:

response_data *<<in>>* *(p. 200)* Response data sent to **DDSDataWriter** (p. 1113) upon acknowledgment

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.10 virtual DDS_ReturnCode_t DDSDataReader::get_matched_publications (DDS_InstanceHandleSeq & *publication_handles*) [inline, virtual]

Retrieve the list of publications currently "associated" with this **DDSDataReader** (p. 1087).

Matching publications are those in the same domain that have a matching **DDSTopic** (p. 1419), compatible QoS common partition that the **DDSDomainParticipant** (p. 1139) has not indicated should be "ignored" by means of the **DDSDomainParticipant::ignore_publication** (p. 1189) operation.

The handles returned in the `publication_handles'` list are the ones that are used by the DDS implementation to locally identify the corresponding matched **DDSDataWriter** (p. 1113) entities. These handles match the ones that appear

in the `instance_handle` field of the **DDS_SampleInfo** (p. 912) when reading the **DDS_PUBLICATION_TOPIC_NAME** (p. 289) builtin topic

Parameters:

publication_handles *<<inout>> (p. 200).* The sequence will be grown if the sequence has ownership and the system has the corresponding resources. Use a sequence without ownership to avoid dynamic memory allocation. If the sequence is too small to store all the matches and the system can not resize the sequence, this method will fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

The maximum number of matches possible is configured with **DDS_DomainParticipantResourceLimitsQosPolicy** (p. 593). You can use a zero-maximum sequence without ownership to quickly check whether there are any matches without allocating any memory. .

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) if the sequence is too small and the system can not resize it, or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.178.2.11 virtual DDS_ReturnCode_t DDSDataReader::get_matched_publication_data (DDS_PublicationBuiltinTopicData & *publication_data*, const DDS_InstanceHandle_t & *publication_handle*) [inline, virtual]

This operation retrieves the information on a publication that is currently "associated" with the **DDSDataReader** (p. 1087).

Publication with a matching **DDSTopic** (p. 1419), compatible QoS and common partition that the application has not indicated should be "ignored" by means of the **DDSDomainParticipant::ignore_publication** (p. 1189) operation.

The `publication_handle` must correspond to a publication currently associated with the **DDSDataReader** (p. 1087). Otherwise, the operation will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315). Use the operation **DDSDataReader::get_matched_publications** (p. 1096) to find the publications that are currently matched with the **DDSDataReader** (p. 1087).

Note: This operation does not retrieve the following information in **DDS_PublicationBuiltinTopicData** (p. 839):

- **DDS_PublicationBuiltinTopicData::type_code** (p. 844)

- **DDS_PublicationBuiltinTopicData::property** (p. 845)

The above information is available through **DDSDataReaderListener::on_data_available()** (p. 1110) (if a reader listener is installed on the **DDSPublicationBuiltinTopicDataDataReader** (p. 1344)).

Parameters:

publication_data *<<inout>> (p. 200).* The information to be filled in on the associated publication. Cannot be NULL.

publication_handle *<<in>> (p. 200).* Handle to a specific publication associated with the **DDSDataWriter** (p. 1113). . Must correspond to a publication currently associated with the **DDSDataReader** (p. 1087).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.178.2.12 virtual DDSTopicDescription* DDSDataReader::get_topicdescription () [inline, virtual]

Returns the **DDSTopicDescription** (p. 1427) associated with the **DDSDataReader** (p. 1087).

Returns that same **DDSTopicDescription** (p. 1427) that was used to create the **DDSDataReader** (p. 1087).

Returns:

DDSTopicDescription (p. 1427) associated with the **DDSDataReader** (p. 1087).

6.178.2.13 virtual DDSSubscriber* DDSDataReader::get_subscriber () [inline, virtual]

Returns the **DDSSubscriber** (p. 1390) to which the **DDSDataReader** (p. 1087) belongs.

Returns:

DDSSubscriber (p. 1390) to which the **DDSDataReader** (p. 1087) belongs.

6.178.2.14 virtual DDS_ReturnCode_t DDSDataReader::get_sample_rejected_status (DDS_SampleRejectedStatus & *status*) [inline, virtual]

Accesses the **DDS_SAMPLE_REJECTED_STATUS** (p. 324) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_SampleRejectedStatus** (p. 925) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.15 virtual DDS_ReturnCode_t DDSDataReader::get_liveliness_changed_status (DDS_LivelinessChangedStatus & *status*) [inline, virtual]

Accesses the **DDS_LIVELINESS_CHANGED_STATUS** (p. 325) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_LivelinessChangedStatus** (p. 775) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.16 virtual DDS_ReturnCode_t DDSDataReader::get_requested_deadline_missed_status (DDS_RequestedDeadlineMissedStatus & *status*) [inline, virtual]

Accesses the **DDS_REQUESTED_DEADLINE_MISSED_STATUS** (p. 323) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_RequestedDeadlineMissedStatus** (p. 875) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.17 virtual DDS_ReturnCode_t DDSDataReader::get_requested_incompatible_qos_status (DDS_RequestedIncompatibleQosStatus & *status*) [inline, virtual]

Accesses the **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_RequestedIncompatibleQosStatus** (p. 877) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.18 virtual DDS_ReturnCode_t DDSDataReader::get_sample_lost_status (DDS_SampleLostStatus & *status*) [inline, virtual]

Accesses the **DDS_SAMPLE_LOST_STATUS** (p. 324) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_SampleLostStatus** (p. 923) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.19 virtual DDS_ReturnCode_t DDSDataReader::get_subscription_matched_status (DDS_SubscriptionMatchedStatus & *status*) [inline, virtual]

Accesses the **DDS_SUBSCRIPTION_MATCHED_STATUS** (p. 326) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_SubscriptionMatchedStatus** (p. 945) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.178.2.20 virtual DDS_ReturnCode_t DDSDataReader::get_datareader_cache_status (DDS_DataReaderCacheStatus & *status*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the datareader cache status for this reader.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_DataReaderCacheStatus** (p. 500) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.178.2.21 virtual DDS_ReturnCode_t DDSDataReader::get_datareader_protocol_status (DDS_DataReaderProtocolStatus & *status*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the datareader protocol status for this reader.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_DataReaderProtocolStatus** (p. 505) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.178.2.22 virtual DDS_ReturnCode_t DDSDataReader::get_matched_publication_datareader_protocol_status (DDS_DataReaderProtocolStatus & *status*, const DDS_InstanceHandle_t & *publication_handle*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the datareader protocol status for this reader, per matched publication identified by the publication_handle.

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted.

Parameters:

status *<<inout>>* *(p. 200)*. The information to be filled in on the associated publication. Cannot be NULL.

publication_handle *<<in>>* *(p. 200)*. Handle to a specific publication associated with the **DDSDataWriter** (p. 1113). . Must correspond to a publication currently associated with the **DDSDataReader** (p. 1087).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.178.2.23 virtual DDS_ReturnCode_t DDSDataReader::set_qos (const DDS_DataReaderQos & *qos*) [inline, virtual]

Sets the reader QoS.

This operation modifies the QoS of the **DDSDataReader** (p. 1087).

The **DDS_DataReaderQos::user_data** (p. 518), **DDS_DataReaderQos::deadline** (p. 517), **DDS_DataReaderQos::latency_budget** (p. 518), **DDS_DataReaderQos::time_based_filter** (p. 519), **DDS_DataReaderQos::reader_data_lifecycle** (p. 519) can be changed. The other policies are immutable.

Parameters:

qos *<<in>>* *(p. 200)* The **DDS_DataReaderQos** (p. 515) to be set to. Policies must be consistent. Immutable policies cannot be changed after **DDSDataReader** (p. 1087) is enabled. The special value **DDS_DATAREADER_QOS_DEFAULT** (p. 99) can be used to indicate that the QoS of the **DDSDataReader** (p. 1087) should be changed to match the current default **DDS_DataReaderQos** (p. 515) set in the **DDSSubscriber** (p. 1390).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_DataReaderQos (p. 515) for rules on consistency among QoS
set_qos (abstract) (p. 1254)
DDSDataReader::set_qos (p. 1102)
Operations Allowed in Listener Callbacks (p. 1320)

6.178.2.24 virtual DDS_ReturnCode_t DDSDataReader::get_qos (DDS_DataReaderQos & *qos*) [inline, virtual]

Gets the reader QoS.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos *<<inout>> (p. 200)* The **DDS_DataReaderQos** (p. 515) to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

get_qos (abstract) (p. 1255)

6.178.2.25 virtual DDS_ReturnCode_t DDSDataReader::set_qos_with_profile (const char * *library_name*, const char * *profile_name*) [inline, virtual]

<<eXtension>> (p. 199) Change the QoS of this reader using the input XML QoS profile.

This operation modifies the QoS of the **DDSDataReader** (p. 1087).

The **DDS_DataReaderQos::user_data** (p. 518), **DDS_DataReaderQos::deadline** (p. 517), **DDS_DataReaderQos::latency_budget** (p. 518), **DDS_DataReaderQos::time_based_filter** (p. 519), **DDS_DataReaderQos::reader_data_lifecycle** (p. 519) can be changed. The other policies are immutable.

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSSubscriber::set_default_library** (p. 1397)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSSubscriber::set_default_profile** (p. 1398)).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_DataReaderQos (p. 515) for rules on consistency among QoS
DDSDataReader::set_qos (p. 1102)
Operations Allowed in Listener Callbacks (p. 1320)

6.178.2.26 virtual DDS_ReturnCode_t DDSDataReader::set_listener (DDSDataReaderListener * *l*, DDS_StatusMask *mask* = DDS_STATUS_MASK_ALL) [inline, virtual]

Sets the reader listener.

Parameters:

l *<<in>>* *(p. 200)* **DDSDataReaderListener** (p. 1108) to set to

mask *<<in>>* *(p. 200)* **DDS_StatusMask** (p. 321) associated with the **DDSDataReaderListener** (p. 1108).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

set_listener (abstract) (p. 1255)

6.178.2.27 virtual DDSDataReaderListener* DDSDataReader::get_listener () [inline, virtual]

Get the reader listener.

Returns:

DDSDataReaderListener (p. 1108) of the **DDSDataReader** (p. 1087).

See also:

get_listener (abstract) (p. 1256)

6.178.2.28 virtual DDS_ReturnCode_t DDSDataReader::enable () [inline, virtual]

Enables the **DDSEntity** (p. 1253).

This operation enables the Entity. Entity objects can be created either enabled or disabled. This is controlled by the value of the **ENTITY_FACTORY** (p. 377) QoS policy on the corresponding factory for the **DDSEntity** (p. 1253).

By default, **ENTITY_FACTORY** (p. 377) is set so that it is not necessary to explicitly call **DDSEntity::enable** (p. 1256) on newly created entities.

The **DDSEntity::enable** (p. 1256) operation is idempotent. Calling enable on an already enabled Entity returns OK and has no effect.

If a **DDSEntity** (p. 1253) has not yet been enabled, the following kinds of operations may be invoked on it:

- set or get the QoS policies (including default QoS policies) and listener
- **DDSEntity::get_statuscondition** (p. 1257)
- 'factory' operations
- **DDSEntity::get_status_changes** (p. 1257) and other get status operations (although the status of a disabled entity never changes)
- 'lookup' operations

Other operations may explicitly state that they may be called on disabled entities; those that do not will return the error **DDS_RETCODE_NOT_ENABLED** (p. 315).

It is legal to delete an **DDSEntity** (p. 1253) that has not been enabled by calling the proper operation on its factory.

Entities created from a factory that is disabled are created disabled, regardless of the setting of the **DDS_EntityFactoryQosPolicy** (p. 733).

Calling enable on an Entity whose factory is not enabled will fail and return **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

If **DDS_EntityFactoryQosPolicy::autoenable_created_entities** (p. 734) is TRUE, the enable operation on a factory will automatically enable all entities created from that factory.

Listeners associated with an entity are not called until the entity is enabled.

Conditions associated with a disabled entity are "inactive," that is, they have a `trigger_value` == FALSE.

Returns:

One of the **Standard Return Codes** (p. 314), **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Implements **DDSEntity** (p. 1256).

6.178.2.29 virtual DDSStatusCondition* DDSDataReader::get_statuscondition () [inline, virtual]

Allows access to the **DDSStatusCondition** (p. 1376) associated with the **DDSEntity** (p. 1253).

The returned condition can then be added to a **DDSWaitSet** (p. 1433) so that the application can wait for specific status changes that affect the **DDSEntity** (p. 1253).

Returns:

the status condition associated with this entity.

Implements **DDSEntity** (p. 1257).

6.178.2.30 virtual DDS_StatusMask DDSDataReader::get_status_changes () [inline, virtual]

Retrieves the list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.

That is, the list of statuses whose value has changed since the last time the application read the status using the get_*_status() method.

When the entity is first created or if the entity is not enabled, all communication statuses are in the "untriggered" state so the list returned by the get_status_changes operation will be empty.

The list of statuses returned by the get_status_changes operation refers to the status that are triggered on the Entity itself and does not include statuses that apply to contained entities.

Returns:

list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.

See also:

Status Kinds (p. 317)

Implements **DDSEntity** (p. 1257).

6.178.2.31 virtual DDS_InstanceHandle_t DDSDataReader::get_instance_handle () [inline, virtual]

Allows access to the **DDS_InstanceHandle_t** (p. 53) associated with the **DDSEntity** (p. 1253).

This operation returns the **DDS_InstanceHandle_t** (p. 53) that represents the **DDSEntity** (p. 1253).

Returns:

the instance handle associated with this entity.

Implements **DDSEntity** (p. 1258).

6.179 DDSDataReaderListener Class Reference

<<*interface*>> *(p. 199)* **DDSListener** (p. 1318) for reader status.

Inheritance diagram for DDSDataReaderListener::

Public Member Functions

- virtual void **on_requested_deadline_missed** (**DDSDataReader** *reader, const **DDS_RequestedDeadlineMissedStatus** &status)

  *Handles the **DDS_REQUESTED_DEADLINE_MISSED_STATUS** (p. 323) communication status.*

- virtual void **on_liveliness_changed** (**DDSDataReader** *reader, const **DDS_LivelinessChangedStatus** &status)

  *Handles the **DDS_LIVELINESS_CHANGED_STATUS** (p. 325) communication status.*

- virtual void **on_requested_incompatible_qos** (**DDSDataReader** *reader, const **DDS_RequestedIncompatibleQosStatus** &status)

  *Handles the **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323) communication status.*

- virtual void **on_sample_rejected** (**DDSDataReader** *reader, const **DDS_SampleRejectedStatus** &status)

  *Handles the **DDS_SAMPLE_REJECTED_STATUS** (p. 324) communication status.*

- virtual void **on_data_available** (**DDSDataReader** *reader)

  *Handle the **DDS_DATA_AVAILABLE_STATUS** (p. 324) communication status.*

- virtual void **on_sample_lost** (**DDSDataReader** *reader, const **DDS_SampleLostStatus** &status)

  *Handles the **DDS_SAMPLE_LOST_STATUS** (p. 324) communication status.*

- virtual void **on_subscription_matched** (**DDSDataReader** *reader, const **DDS_SubscriptionMatchedStatus** &status)

  *Handles the **DDS_SUBSCRIPTION_MATCHED_STATUS** (p. 326) communication status.*

6.179.1 Detailed Description

<<*interface*>> *(p. 199)* **DDSListener** (p. 1318) for reader status.

Entity:

DDSDataReader (p. 1087)

Status:

DDS_DATA_AVAILABLE_STATUS (p. 324);
DDS_LIVELINESS_CHANGED_STATUS (p. 325), **DDS_LivelinessChangedStatus** (p. 775);
DDS_REQUESTED_DEADLINE_MISSED_STATUS (p. 323), **DDS_RequestedDeadlineMissedStatus** (p. 875);
DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_RequestedIncompatibleQosStatus** (p. 877);
DDS_SAMPLE_LOST_STATUS (p. 324), **DDS_SampleLostStatus** (p. 923);
DDS_SAMPLE_REJECTED_STATUS (p. 324), **DDS_SampleRejectedStatus** (p. 925);
DDS_SUBSCRIPTION_MATCHED_STATUS (p. 326), **DDS_SubscriptionMatchedStatus** (p. 945);

See also:

Status Kinds (p. 317)
Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_subscriber.cxx.

6.179.2 Member Function Documentation

6.179.2.1 virtual void DDSDataReaderListener::on_requested_deadline_missed (DDSDataReader * *reader*, const DDS_RequestedDeadlineMissedStatus & *status*) [virtual]

Handles the **DDS_REQUESTED_DEADLINE_MISSED_STATUS** (p. 323) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.179.2.2 virtual void DDSDataReaderListener::on_liveliness_changed (DDSDataReader * *reader*, const DDS_LivelinessChangedStatus & *status*) [virtual]

Handles the **DDS_LIVELINESS_CHANGED_STATUS** (p. 325) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.179.2.3 virtual void DDSDataReaderListener::on_requested_incompatible_qos (DDSDataReader * *reader*, const DDS_RequestedIncompatibleQosStatus & *status*) [virtual]

Handles the **DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS** (p. 323) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.179.2.4 virtual void DDSDataReaderListener::on_sample_rejected (DDSDataReader * *reader*, const DDS_SampleRejectedStatus & *status*) [virtual]

Handles the **DDS_SAMPLE_REJECTED_STATUS** (p. 324) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.179.2.5 virtual void DDSDataReaderListener::on_data_available (DDSDataReader * *reader*) [virtual]

Handle the **DDS_DATA_AVAILABLE_STATUS** (p. 324) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.179.2.6 virtual void DDSDataReaderListener::on_sample_lost (DDSDataReader * *reader*, const DDS_SampleLostStatus & *status*) [virtual]

Handles the **DDS_SAMPLE_LOST_STATUS** (p. 324) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.179.2.7 virtual void DDSDataReaderListener::on_subscription_matched (DDSDataReader * *reader*, const DDS_SubscriptionMatchedStatus & *status*) [virtual]

Handles the **DDS_SUBSCRIPTION_MATCHED_STATUS** (p. 326) communication status.

Examples:

HelloWorld_subscriber.cxx.

6.180 DDSDataReaderSeq Class Reference

Declares IDL `sequence` < **DDSDataReader** (p. 1087) > .

6.180.1 Detailed Description

Declares IDL `sequence` < **DDSDataReader** (p. 1087) > .

See also:

FooSeq (p. 1494)

6.181 DDSDataWriter Class Reference

<<***interface***>> *(p. 199)* Allows an application to set the value of the data to be published under a given **DDSTopic** (p. 1419).

Inheritance diagram for DDSDataWriter::

Public Member Functions

- virtual **DDS_ReturnCode_t get_liveliness_lost_status** (**DDS_LivelinessLostStatus** &status)

  *Accesses the **DDS_LIVELINESS_LOST_STATUS** (p. 325) communication status.*

- virtual **DDS_ReturnCode_t get_offered_deadline_missed_status** (**DDS_OfferedDeadlineMissedStatus** &status)

  *Accesses the **DDS_OFFERED_DEADLINE_MISSED_STATUS** (p. 323) communication status.*

- virtual **DDS_ReturnCode_t get_offered_incompatible_qos_status** (**DDS_OfferedIncompatibleQosStatus** &status)

  *Accesses the **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323) communication status.*

- virtual **DDS_ReturnCode_t get_publication_matched_status** (**DDS_PublicationMatchedStatus** &status)

  *Accesses the **DDS_PUBLICATION_MATCHED_STATUS** (p. 325) communication status.*

- virtual **DDS_ReturnCode_t get_reliable_writer_cache_changed_status** (**DDS_ReliableWriterCacheChangedStatus** &status)

  <<**eXtension**>> (p. 199) *Get the reliable cache status for this writer.*

- virtual **DDS_ReturnCode_t get_reliable_reader_activity_changed_status** (**DDS_ReliableReaderActivityChangedStatus** &status)

  <<**eXtension**>> (p. 199) *Get the reliable reader activity changed status for this writer.*

- virtual **DDS_ReturnCode_t get_datawriter_cache_status** (**DDS_DataWriterCacheStatus** &status)

  <<**eXtension**>> (p. 199) *Get the datawriter cache status for this writer.*

^ virtual **DDS_ReturnCode_t get_datawriter_protocol_status** (**DDS_DataWriterProtocolStatus** &status)

<<**eXtension**>> (p. 199) *Get the datawriter protocol status for this writer.*

^ virtual **DDS_ReturnCode_t get_matched_subscription_datawriter_protocol_status** (**DDS_DataWriterProtocolStatus** &status, const **DDS_InstanceHandle_t** &subscription_handle)

<<**eXtension**>> (p. 199) *Get the datawriter protocol status for this writer, per matched subscription identified by the subscription_handle.*

^ virtual **DDS_ReturnCode_t get_matched_subscription_datawriter_protocol_status_by_locator** (**DDS_DataWriterProtocolStatus** &status, const **DDS_Locator_t** &locator)

<<**eXtension**>> (p. 199) *Get the datawriter protocol status for this writer, per matched subscription identified by the locator.*

^ virtual **DDS_ReturnCode_t assert_liveliness** ()

*This operation manually asserts the liveliness of this **DDSDataWriter** (p. 1113).*

^ virtual **DDS_ReturnCode_t get_matched_subscription_locators** (**DDS_LocatorSeq** &locators)

<<**eXtension**>> (p. 199) *Retrieve the list of locators for subscriptions currently "associated" with this **DDSDataWriter** (p. 1113).*

^ virtual **DDS_ReturnCode_t get_matched_subscriptions** (**DDS_InstanceHandleSeq** &subscription_handles)

*Retrieve the list of subscriptions currently "associated" with this **DDSDataWriter** (p. 1113).*

^ virtual **DDS_ReturnCode_t get_matched_subscription_data** (**DDS_SubscriptionBuiltinTopicData** &subscription_data, const **DDS_InstanceHandle_t** &subscription_handle)

*This operation retrieves the information on a subscription that is currently "associated" with the **DDSDataWriter** (p. 1113).*

^ virtual **DDSTopic** * **get_topic** ()

*This operation returns the **DDSTopic** (p. 1419) associated with the **DDSDataWriter** (p. 1113).*

^ virtual **DDSPublisher** * **get_publisher** ()

*This operation returns the **DDSPublisher** (p. 1346) to which the **DDSDataWriter** (p. 1113) belongs.*

- virtual **DDS_ReturnCode_t wait_for_acknowledgments** (const **DDS_Duration_t** &max_wait)

  *Blocks the calling thread until all data written by reliable **DDSDataWriter** (p. 1113) entity is acknowledged, or until timeout expires.*

- virtual **DDS_ReturnCode_t wait_for_asynchronous_publishing** (const **DDS_Duration_t** &max_wait)

  <<**eXtension**>> (p. 199) *Blocks the calling thread until asynchronous sending is complete.*

- virtual **DDS_ReturnCode_t set_qos** (const **DDS_DataWriterQos** &qos)

  Sets the writer QoS.

- virtual **DDS_ReturnCode_t set_qos_with_profile** (const char *library_name, const char *profile_name)

  <<**eXtension**>> (p. 199) *Change the QoS of this writer using the input XML QoS profile.*

- virtual **DDS_ReturnCode_t get_qos** (**DDS_DataWriterQos** &qos)

  Gets the writer QoS.

- virtual **DDS_ReturnCode_t set_listener** (**DDSDataWriterListener** *l, **DDS_StatusMask** mask=DDS_STATUS_MASK_ALL)

  Sets the writer listener.

- virtual **DDSDataWriterListener** * **get_listener** ()

  Get the writer listener.

- virtual **DDS_ReturnCode_t flush** ()

  <<**eXtension**>> (p. 199) *Flushes the batch in progress in the context of the calling thread.*

- virtual **DDS_ReturnCode_t enable** ()

  *Enables the **DDSEntity** (p. 1253).*

- virtual **DDSStatusCondition** * **get_statuscondition** ()

  *Allows access to the **DDSStatusCondition** (p. 1376) associated with the **DDSEntity** (p. 1253).*

- virtual **DDS_StatusMask get_status_changes** ()

  *Retrieves the list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.*

ˆ virtual **DDS_InstanceHandle_t get_instance_handle** ()

*Allows access to the **DDS_InstanceHandle_t** (p. 53) associated with the **DDSEntity** (p. 1253).*

6.181.1 Detailed Description

<<***interface***>> *(p. 199)* Allows an application to set the value of the data to be published under a given **DDSTopic** (p. 1419).

QoS:

DDS_DataWriterQos (p. 553)

Status:

DDS_LIVELINESS_LOST_STATUS (p. 325), **DDS_LivelinessLostStatus** (p. 777);
DDS_OFFERED_DEADLINE_MISSED_STATUS (p. 323), **DDS_OfferedDeadlineMissedStatus** (p. 803);
DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_OfferedIncompatibleQosStatus** (p. 805);
DDS_PUBLICATION_MATCHED_STATUS (p. 325), **DDS_PublicationMatchedStatus** (p. 848);
DDS_RELIABLE_READER_ACTIVITY_CHANGED_STATUS (p. 327), **DDS_ReliableReaderActivityChangedStatus** (p. 869);
DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS (p. 326), DDS_ReliableWriterCacheChangedStatus

Listener:

DDSDataWriterListener (p. 1133)

A **DDSDataWriter** (p. 1113) is attached to exactly one **DDSPublisher** (p. 1346), that acts as a factory for it.

A **DDSDataWriter** (p. 1113) is bound to exactly one **DDSTopic** (p. 1419) and therefore to exactly one data type. The **DDSTopic** (p. 1419) must exist prior to the **DDSDataWriter** (p. 1113)'s creation.

DDSDataWriter (p. 1113) is an abstract class. It must be specialized for each particular application data-type (see **USER_DATA** (p. 345)). The additional methods or functions that must be defined in the auto-generated class for a hypothetical application type `Foo` (p. 1443) are specified in the example type **DDSDataWriter** (p. 1113).

The following operations may be called even if the **DDSDataWriter** (p. 1113) is not enabled. Other operations will fail with **DDS_RETCODE_NOT_ENABLED** (p. 315) if called on a disabled **DDSDataWriter** (p. 1113):

- The base-class operations **DDSDataWriter::set_qos** (p. 1126), **DDSDataWriter::get_qos** (p. 1128), **DDSDataWriter::set_listener** (p. 1128), **DDSDataWriter::get_listener** (p. 1129), **DDSEntity::enable** (p. 1256), **DDSEntity::get_statuscondition** (p. 1257) and **DDSEntity::get_status_changes** (p. 1257)
- **DDSDataWriter::get_liveliness_lost_status** (p. 1117) **DDSDataWriter::get_offered_deadline_missed_status** (p. 1118) **DDSDataWriter::get_offered_incompatible_qos_status** (p. 1118) **DDSDataWriter::get_publication_matched_status** (p. 1118) **DDSDataWriter::get_reliable_writer_cache_changed_status** (p. 1119) **DDSDataWriter::get_reliable_reader_activity_changed_status** (p. 1119)

Several **DDSDataWriter** (p. 1113) may operate in different threads. If they share the same **DDSPublisher** (p. 1346), the middleware guarantees that its operations are thread-safe.

See also:

FooDataWriter (p. 1475)
Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_publisher.cxx, and **HelloWorldSupport.cxx**.

6.181.2 Member Function Documentation

6.181.2.1 virtual DDS_ReturnCode_t DDSDataWriter::get_liveliness_lost_status (DDS_LivelinessLostStatus & *status*) [inline, virtual]

Accesses the **DDS_LIVELINESS_LOST_STATUS** (p. 325) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_LivelinessLostStatus** (p. 777) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.181.2.2 virtual DDS_ReturnCode_t DDSDataWriter::get_offered_deadline_missed_status (DDS_OfferedDeadlineMissedStatus & *status*) [inline, virtual]

Accesses the **DDS_OFFERED_DEADLINE_MISSED_STATUS** (p. 323) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_OfferedDeadlineMissedStatus** (p. 803) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.181.2.3 virtual DDS_ReturnCode_t DDSDataWriter::get_offered_incompatible_qos_status (DDS_OfferedIncompatibleQosStatus & *status*) [inline, virtual]

Accesses the **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_OfferedIncompatibleQosStatus** (p. 805) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.181.2.4 virtual DDS_ReturnCode_t DDSDataWriter::get_publication_matched_status (DDS_PublicationMatchedStatus & *status*) [inline, virtual]

Accesses the **DDS_PUBLICATION_MATCHED_STATUS** (p. 325) communication status.

Parameters:

status <<*inout*>> *(p. 200)* **DDS_PublicationMatchedStatus** (p. 848) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.181.2.5 virtual DDS_ReturnCode_t DDSDataWriter::get_reliable_writer_cache_changed_status (DDS_ReliableWriterCacheChangedStatus & *status*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the reliable cache status for this writer.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_ReliableWriterCacheChangedStatus** (p. 871) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.181.2.6 virtual DDS_ReturnCode_t DDSDataWriter::get_reliable_reader_activity_changed_status (DDS_ReliableReaderActivityChangedStatus & *status*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the reliable reader activity changed status for this writer.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_ReliableReaderActivityChangedStatus** (p. 869) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.181.2.7 virtual DDS_ReturnCode_t DDSDataWriter::get_datawriter_cache_status (DDS_DataWriterCacheStatus & *status*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the datawriter cache status for this writer.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_DataWriterCacheStatus** (p. 534) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.181.2.8 virtual DDS_ReturnCode_t DDSDataWriter::get_datawriter_protocol_status (DDS_DataWriterProtocolStatus & *status*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the datawriter protocol status for this writer.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_DataWriterProtocolStatus** (p. 540) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.181.2.9 virtual DDS_ReturnCode_t DDSDataWriter::get_matched_subscription_datawriter_protocol_status (DDS_DataWriterProtocolStatus & *status*, const DDS_InstanceHandle_t & *subscription_handle*) [inline, virtual]

<<eXtension>> *(p. 199)* Get the datawriter protocol status for this writer, per matched subscription identified by the subscription_handle.

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted.

Parameters:

status *<<inout>>* *(p. 200)* **DDS_DataWriterProtocolStatus** (p. 540) to be filled in.

subscription_handle *<<in>>* *(p. 200)* Handle to a specific subscription associated with the **DDSDataReader** (p. 1087). Must correspond to a subscription currently associated with the **DDSDataWriter** (p. 1113).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.181.2.10 virtual DDS_ReturnCode_t DDSDataWriter::get_matched_subscription_datawriter_protocol_status_by_locator (DDS_DataWriterProtocolStatus & *status*, const DDS_Locator_t & *locator*) [inline, virtual]

<<eXtension>> (p. *199*) Get the datawriter protocol status for this writer, per matched subscription identified by the locator.

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted.

Parameters:

status *<<inout>>* (p. *200*) **DDS_DataWriterProtocolStatus** (p. 540) to be filled in

locator *<<in>>* (p. *200*) Locator to a specific locator associated with the **DDSDataReader** (p. 1087). Must correspond to a locator of one or more subscriptions currently associated with the **DDSDataWriter** (p. 1113).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.181.2.11 virtual DDS_ReturnCode_t DDSDataWriter::assert_liveliness () [inline, virtual]

This operation manually asserts the liveliness of this **DDSDataWriter** (p. 1113).

This is used in combination with the **LIVELINESS** (p. 358) policy to indicate to RTI Connext that the **DDSDataWriter** (p. 1113) remains active.

You only need to use this operation if the **LIVELINESS** (p. 358) setting is either **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** (p. 359) or **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS** (p. 359). Otherwise, it has no effect.

Note: writing data via the **FooDataWriter::write** (p. 1484) or **FooDataWriter::write_w_timestamp** (p. 1486) operation asserts liveliness on the

DDSDataWriter (p. 1113) itself, and its **DDSDomainParticipant** (p. 1139). Consequently the use of **assert_liveliness()** (p. 1121) is only needed if the application is not writing data regularly.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_LivelinessQosPolicy (p. 779)

6.181.2.12 virtual DDS_ReturnCode_t DDSDataWriter::get_matched_subscription_locators (DDS_LocatorSeq & *locators*) [inline, virtual]

<<eXtension>> (p. 199) Retrieve the list of locators for subscriptions currently "associated" with this **DDSDataWriter** (p. 1113).

Matched subscription locators include locators for all those subscriptions in the same domain that have a matching **DDSTopic** (p. 1419), compatible QoS and common partition that the **DDSDomainParticipant** (p. 1139) has not indicated should be "ignored" by means of the **DDSDomainParticipant::ignore_subscription** (p. 1190) operation.

The locators returned in the `locators` list are the ones that are used by the DDS implementation to communicate with the corresponding matched **DDSDataReader** (p. 1087) entities.

Parameters:

locators *<<inout>> (p. 200).* Handles of all the matched subscription locators.

The sequence will be grown if the sequence has ownership and the system has the corresponding resources. Use a sequence without ownership to avoid dynamic memory allocation. If the sequence is too small to store all the matches and the system can not resize the sequence, this method will fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315). .

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) if the sequence is too small and the system can not resize it, or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.181.2.13 virtual DDS_ReturnCode_t DDSDataWriter::get_matched_subscriptions (DDS_InstanceHandleSeq & *subscription_handles*) [inline, virtual]

Retrieve the list of subscriptions currently "associated" with this **DDSDataWriter** (p. 1113).

Matched subscriptions include all those in the same domain that have a matching **DDSTopic** (p. 1419), compatible QoS and common partition that the **DDSDomainParticipant** (p. 1139) has not indicated should be "ignored" by means of the **DDSDomainParticipant::ignore_subscription** (p. 1190) operation.

The handles returned in the `subscription_handles` list are the ones that are used by the DDS implementation to locally identify the corresponding matched **DDSDataReader** (p. 1087) entities. These handles match the ones that appear in the **DDS_SampleInfo::instance_handle** (p. 917) field of the **DDS_SampleInfo** (p. 912) when reading the **DDS_SUBSCRIPTION_TOPIC_NAME** (p. 291) builtin topic.

Parameters:

subscription_handles *<<inout>> (p. 200).* Handles of all the matched subscriptions.

The sequence will be grown if the sequence has ownership and the system has the corresponding resources. Use a sequence without ownership to avoid dynamic memory allocation. If the sequence is too small to store all the matches and the system can not resize the sequence, this method will fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

The maximum number of matches possible is configured with **DDS_DomainParticipantResourceLimitsQosPolicy** (p. 593). You can use a zero-maximum sequence without ownership to quickly check whether there are any matches without allocating any memory. .

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) if the sequence is too small and the system can not resize it, or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.181.2.14 virtual DDS_ReturnCode_t DDSDataWriter::get_matched_subscription_data (DDS_SubscriptionBuiltinTopicData & *subscription_data*, const DDS_InstanceHandle_t & *subscription_handle*) [inline, virtual]

This operation retrieves the information on a subscription that is currently "associated" with the **DDSDataWriter** (p. 1113).

The `subscription_handle` must correspond to a subscription currently associated with the **DDSDataWriter** (p. 1113). Otherwise, the operation will fail and fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315). Use **DDSDataWriter::get_matched_subscriptions** (p. 1123) to find the subscriptions that are currently matched with the **DDSDataWriter** (p. 1113).

Note: This operation does not retrieve the following information in **DDS_SubscriptionBuiltinTopicData** (p. 936):

- **DDS_SubscriptionBuiltinTopicData::type_code** (p. 941)
- **DDS_SubscriptionBuiltinTopicData::property** (p. 942)
- **DDS_SubscriptionBuiltinTopicData::content_filter_property** (p. 942)

The above information is available through **DDSDataReaderListener::on_data_available()** (p. 1110) (if a reader listener is installed on the **DDSSubscriptionBuiltinTopicDataDataReader** (p. 1417)).

Parameters:

subscription_data *<<inout>> (p. 200).* The information to be filled in on the associated subscription. Cannot be NULL.

subscription_handle *<<in>> (p. 200).* Handle to a specific subscription associated with the **DDSDataReader** (p. 1087). . Must correspond to a subscription currently associated with the **DDSDataWriter** (p. 1113).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.181.2.15 virtual DDSTopic* DDSDataWriter::get_topic () [inline, virtual]

This operation returns the **DDSTopic** (p. 1419) associated with the **DDSDataWriter** (p. 1113).

This is the same **DDSTopic** (p. 1419) that was used to create the **DDSDataWriter** (p. 1113).

Returns:

DDSTopic (p. 1419) that was used to create the **DDSDataWriter** (p. 1113).

6.181.2.16 virtual DDSPublisher* DDSDataWriter::get_publisher () [inline, virtual]

This operation returns the **DDSPublisher** (p. 1346) to which the **DDSDataWriter** (p. 1113) belongs.

Returns:

DDSPublisher (p. 1346) to which the **DDSDataWriter** (p. 1113) belongs.

6.181.2.17 virtual DDS_ReturnCode_t DDSDataWriter::wait_for_acknowledgments (const DDS_Duration_t & *max_wait*) [inline, virtual]

Blocks the calling thread until all data written by reliable **DDSDataWriter** (p. 1113) entity is acknowledged, or until timeout expires.

This operation blocks the calling thread until either all data written by the reliable **DDSDataWriter** (p. 1113) entity is acknowledged by (a) all reliable **DDSDataReader** (p. 1087) entities that are matched and alive and (b) by all required subscriptions, or until the duration specified by the `max_wait` parameter elapses, whichever happens first. A successful completion indicates that all the samples written have been acknowledged by all reliable matched data readers and by all required subscriptions; a return value of TIMEOUT indicates that max_wait elapsed before all the data was acknowledged.

Note that if a thread is blocked in the call to wait_for_acknowledgments on a DataWriter and a different thread writes new samples on the same DataWriter, the new samples must be acknowledged before unblocking the thread waiting on wait_for_acknowledgments.

If the **DDSDataWriter** (p. 1113) does not have **DDS_ReliabilityQosPolicy** (p. 865) kind set to RELIABLE, this operation will complete immediately with RETCODE_OK.

Parameters:

max_wait *<<in>> (p. 200)* Specifies maximum time to wait for acknowledgements **DDS_Duration_t** (p. 621) .

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_NOT_ENABLED** (p. 315), **DDS_RETCODE_TIMEOUT** (p. 316)

6.181.2.18 virtual DDS_ReturnCode_t DDSDataWriter::wait_for_asynchronous_publishing (const DDS_Duration_t & *max_wait*) [inline, virtual]

<<eXtension>> *(p. 199)* Blocks the calling thread until asynchronous sending is complete.

This operation blocks the calling thread (up to `max_wait`) until all data written by the asynchronous **DDSDataWriter** (p. 1113) is sent and acknowledged (if reliable) by all matched **DDSDataReader** (p. 1087) entities. A successful completion indicates that all the samples written have been sent and acknowledged where applicable; a time out indicates that `max_wait` elapsed before all the data was sent and/or acknowledged.

In other words, this guarantees that sending to best effort **DDSDataReader** (p. 1087) is complete in addition to what **DDSDataWriter::wait_for_acknowledgments** (p. 1125) provides.

If the **DDSDataWriter** (p. 1113) does not have **DDS_PublishModeQosPolicy** (p. 853) kind set to **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) the operation will complete immediately with **DDS_RETCODE_OK** (p. 315).

Parameters:

max_wait *<<in>>* *(p. 200)* Specifies maximum time to wait for acknowledgements **DDS_Duration_t** (p. 621) .

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_NOT_ENABLED** (p. 315), **DDS_RETCODE_TIMEOUT** (p. 316)

6.181.2.19 virtual DDS_ReturnCode_t DDSDataWriter::set_qos (const DDS_DataWriterQos & *qos*) [inline, virtual]

Sets the writer QoS.

This operation modifies the QoS of the **DDSDataWriter** (p. 1113).

The **DDS_DataWriterQos::user_data** (p. 557), **DDS_DataWriterQos::deadline** (p. 556), **DDS_DataWriterQos::latency_budget** (p. 556), **DDS_DataWriterQos::ownership_strength**

(p. 557), **DDS_DataWriterQos::transport_priority** (p. 557), **DDS_DataWriterQos::lifespan** (p. 557) and **DDS_DataWriterQos::writer_data_lifecycle** (p. 557) can be changed. The other policies are immutable.

Parameters:

qos <<*in*>> *(p. 200)* The **DDS_DataWriterQos** (p. 553) to be set to. Policies must be consistent. Immutable policies cannot be changed after **DDSDataWriter** (p. 1113) is enabled. The special value **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) can be used to indicate that the QoS of the **DDSDataWriter** (p. 1113) should be changed to match the current default **DDS_DataWriterQos** (p. 553) set in the **DDSPublisher** (p. 1346).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_DataWriterQos (p. 553) for rules on consistency among QoS
set_qos (abstract) (p. 1254)
Operations Allowed in Listener Callbacks (p. 1320)

6.181.2.20 virtual DDS_ReturnCode_t DDSDataWriter::set_qos_with_profile (const char * *library_name*, const char * *profile_name*) [inline, virtual]

<<*eXtension*>> *(p. 199)* Change the QoS of this writer using the input XML QoS profile.

This operation modifies the QoS of the **DDSDataWriter** (p. 1113).

The **DDS_DataWriterQos::user_data** (p. 557), **DDS_DataWriterQos::deadline** (p. 556), **DDS_DataWriterQos::latency_budget** (p. 556), **DDS_DataWriterQos::ownership_strength** (p. 557), **DDS_DataWriterQos::transport_priority** (p. 557), **DDS_DataWriterQos::lifespan** (p. 557) and **DDS_DataWriterQos::writer_data_lifecycle** (p. 557) can be changed. The other policies are immutable.

Parameters:

library_name <<*in*>> *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSPublisher::set_default_library** (p. 1352)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSPublisher::set_default_profile** (p. 1353)).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_DataWriterQos (p. 553) for rules on consistency among QoS
Operations Allowed in Listener Callbacks (p. 1320)

6.181.2.21 virtual DDS_ReturnCode_t DDSDataWriter::get_qos (DDS_DataWriterQos & *qos*) [inline, virtual]

Gets the writer QoS.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos *<<inout>>* *(p. 200)* The **DDS_DataWriterQos** (p. 553) to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

get_qos (abstract) (p. 1255)

6.181.2.22 virtual DDS_ReturnCode_t DDSDataWriter::set_listener (DDSDataWriterListener * *l*, DDS_StatusMask *mask* = DDS_STATUS_MASK_ALL) [inline, virtual]

Sets the writer listener.

Parameters:

l *<<in>>* *(p. 200)* **DDSDataWriterListener** (p. 1133) to set to

mask *<<in>> (p. 200)* **DDS_StatusMask** (p. 321) associated with the **DDSDataWriterListener** (p. 1133).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

set_listener (abstract) (p. 1255)

6.181.2.23 virtual DDSDataWriterListener* DDSDataWriter::get_listener () [inline, virtual]

Get the writer listener.

Returns:

DDSDataWriterListener (p. 1133) of the **DDSDataWriter** (p. 1113).

See also:

get_listener (abstract) (p. 1256)

6.181.2.24 virtual DDS_ReturnCode_t DDSDataWriter::flush () [inline, virtual]

*<<**eXtension**>> (p. 199)* Flushes the batch in progress in the context of the calling thread.

After being flushed, the batch is available to be sent on the network.

If the **DDSDataWriter** (p. 1113) does not have **DDS_PublishModeQosPolicy** (p. 853) kind set to **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422), the batch will be sent on the network immediately (in the context of the calling thread).

If the **DDSDataWriter** (p. 1113) does have **DDS_PublishModeQosPolicy** (p. 853) kind set to **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422), the batch will be sent in the context of the asynchronous publishing thread.

This operation may block in the same conditions as **FooDataWriter::write** (p. 1484).

If this operation does block, the RELIABILITY max_blocking_time configures the maximum time the write operation may block (waiting for space to become

available). If max_blocking_time elapses before the DDS_DataWriter is able to store the modification without exceeding the limits, the operation will fail with DDS_RETCODE_TIMEOUT.

MT Safety:

flush() (p. 1129) is only thread-safe with batching if **DDS_BatchQosPolicy::thread_safe_write** (p. 479) is TRUE.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.181.2.25 virtual DDS_ReturnCode_t DDSDataWriter::enable () [inline, virtual]

Enables the **DDSEntity** (p. 1253).

This operation enables the Entity. Entity objects can be created either enabled or disabled. This is controlled by the value of the **ENTITY_FACTORY** (p. 377) QoS policy on the corresponding factory for the **DDSEntity** (p. 1253).

By default, **ENTITY_FACTORY** (p. 377) is set so that it is not necessary to explicitly call **DDSEntity::enable** (p. 1256) on newly created entities.

The **DDSEntity::enable** (p. 1256) operation is idempotent. Calling enable on an already enabled Entity returns OK and has no effect.

If a **DDSEntity** (p. 1253) has not yet been enabled, the following kinds of operations may be invoked on it:

- set or get the QoS policies (including default QoS policies) and listener
- **DDSEntity::get_statuscondition** (p. 1257)
- 'factory' operations
- **DDSEntity::get_status_changes** (p. 1257) and other get status operations (although the status of a disabled entity never changes)
- 'lookup' operations

Other operations may explicitly state that they may be called on disabled entities; those that do not will return the error **DDS_RETCODE_NOT_ENABLED** (p. 315).

It is legal to delete an **DDSEntity** (p. 1253) that has not been enabled by calling the proper operation on its factory.

Entities created from a factory that is disabled are created disabled, regardless of the setting of the **DDS_EntityFactoryQosPolicy** (p. 733).

Calling enable on an Entity whose factory is not enabled will fail and return **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

If **DDS_EntityFactoryQosPolicy::autoenable_created_entities** (p. 734) is TRUE, the enable operation on a factory will automatically enable all entities created from that factory.

Listeners associated with an entity are not called until the entity is enabled.

Conditions associated with a disabled entity are "inactive," that is, they have a `trigger_value` == FALSE.

Returns:

One of the **Standard Return Codes** (p. 314), **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Implements **DDSEntity** (p. 1256).

6.181.2.26 virtual DDSStatusCondition* DDSDataWriter::get_statuscondition () [inline, virtual]

Allows access to the **DDSStatusCondition** (p. 1376) associated with the **DDSEntity** (p. 1253).

The returned condition can then be added to a **DDSWaitSet** (p. 1433) so that the application can wait for specific status changes that affect the **DDSEntity** (p. 1253).

Returns:

the status condition associated with this entity.

Implements **DDSEntity** (p. 1257).

6.181.2.27 virtual DDS_StatusMask DDSDataWriter::get_status_changes () [inline, virtual]

Retrieves the list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.

That is, the list of statuses whose value has changed since the last time the application read the status using the get_*_status() method.

When the entity is first created or if the entity is not enabled, all communication statuses are in the "untriggered" state so the list returned by the get_status_changes operation will be empty.

The list of statuses returned by the get_status_changes operation refers to the status that are triggered on the Entity itself and does not include statuses that apply to contained entities.

Returns:

list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.

See also:

Status Kinds (p. 317)

Implements **DDSEntity** (p. 1257).

6.181.2.28 virtual DDS_InstanceHandle_t DDSDataWriter::get_instance_handle () [inline, virtual]

Allows access to the **DDS_InstanceHandle_t** (p. 53) associated with the **DDSEntity** (p. 1253).

This operation returns the **DDS_InstanceHandle_t** (p. 53) that represents the **DDSEntity** (p. 1253).

Returns:

the instance handle associated with this entity.

Implements **DDSEntity** (p. 1258).

6.182 DDSDataWriterListener Class Reference

<<*interface*>> *(p. 199)* **DDSListener** (p. 1318) for writer status.

Inheritance diagram for DDSDataWriterListener::

Public Member Functions

- virtual void **on_offered_deadline_missed** (**DDSDataWriter** *writer, const **DDS_OfferedDeadlineMissedStatus** &status)

  *Handles the **DDS_OFFERED_DEADLINE_MISSED_STATUS** (p. 323) status.*

- virtual void **on_liveliness_lost** (**DDSDataWriter** *writer, const **DDS_LivelinessLostStatus** &status)

  *Handles the **DDS_LIVELINESS_LOST_STATUS** (p. 325) status.*

- virtual void **on_offered_incompatible_qos** (**DDSDataWriter** *writer, const **DDS_OfferedIncompatibleQosStatus** &status)

  *Handles the **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323) status.*

- virtual void **on_publication_matched** (**DDSDataWriter** *writer, const **DDS_PublicationMatchedStatus** &status)

  *Handles the **DDS_PUBLICATION_MATCHED_STATUS** (p. 325) status.*

- virtual void **on_reliable_writer_cache_changed** (**DDSDataWriter** *writer, const **DDS_ReliableWriterCacheChangedStatus** &status)

  <<**eXtension**>> (p. 199) *A change has occurred in the writer's cache of unacknowledged samples.*

- virtual void **on_reliable_reader_activity_changed** (**DDSDataWriter** *writer, const **DDS_ReliableReaderActivityChangedStatus** &status)

  <<**eXtension**>> (p. 199) *A matched reliable reader has become active or become inactive.*

- virtual void **on_instance_replaced** (**DDSDataWriter** *writer, const **DDS_InstanceHandle_t** &handle)

  Notifies when an instance is replaced in DataWriter queue.

6.182.1 Detailed Description

<<*interface*>> *(p. 199)* **DDSListener** (p. 1318) for writer status.

Entity:

DDSDataWriter (p. 1113)

Status:

DDS_LIVELINESS_LOST_STATUS (p. 325), **DDS_LivelinessLostStatus** (p. 777);
DDS_OFFERED_DEADLINE_MISSED_STATUS (p. 323), **DDS_OfferedDeadlineMissedStatus** (p. 803);
DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_OfferedIncompatibleQosStatus** (p. 805);
DDS_PUBLICATION_MATCHED_STATUS (p. 325), **DDS_PublicationMatchedStatus** (p. 848);
DDS_RELIABLE_READER_ACTIVITY_CHANGED_STATUS (p. 327), **DDS_ReliableReaderActivityChangedStatus** (p. 869);
DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS (p. 326), **DDS_ReliableWriterCacheChangedStatus** (p. 871);

See also:

Status Kinds (p. 317)
Operations Allowed in Listener Callbacks (p. 1320)

6.182.2 Member Function Documentation

6.182.2.1 virtual void DDSDataWriterListener::on_offered_deadline_missed (DDSDataWriter * *writer*, const DDS_OfferedDeadlineMissedStatus & *status*) [virtual]

Handles the **DDS_OFFERED_DEADLINE_MISSED_STATUS** (p. 323) status.

This callback is called when the deadline that the **DDSDataWriter** (p. 1113) has committed through its **DEADLINE** (p. 353) qos policy was not respected for a specific instance. This callback is called for each deadline period elapsed during which the **DDSDataWriter** (p. 1113) failed to provide data for an instance.

Parameters:

writer <<*out*>> *(p. 200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

status *<<out>>* *(p. 200)* Current deadline missed status of locally created **DDSDataWriter** (p. 1113)

6.182.2.2 virtual void DDSDataWriterListener::on_liveliness_lost (DDSDataWriter * *writer*, const DDS_LivelinessLostStatus & *status*) [virtual]

Handles the **DDS_LIVELINESS_LOST_STATUS** (p. 325) status.

This callback is called when the liveliness that the **DDSDataWriter** (p. 1113) has committed through its **LIVELINESS** (p. 358) qos policy was not respected; this **DDSDataReader** (p. 1087) entities will consider the **DDSDataWriter** (p. 1113) as no longer "alive/active". This callback will not be called when an already not alive **DDSDataWriter** (p. 1113) simply renames not alive for another liveliness period.

Parameters:

writer *<<out>>* *(p. 200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

status *<<out>>* *(p. 200)* Current liveliness lost status of locally created **DDSDataWriter** (p. 1113)

6.182.2.3 virtual void DDSDataWriterListener::on_offered_incompatible_qos (DDSDataWriter * *writer*, const DDS_OfferedIncompatibleQosStatus & *status*) [virtual]

Handles the **DDS_OFFERED_INCOMPATIBLE_QOS_STATUS** (p. 323) status.

This callback is called when the **DDS_DataWriterQos** (p. 553) of the **DDSDataWriter** (p. 1113) was incompatible with what was requested by a **DDSDataReader** (p. 1087). This callback is called when a **DDSDataWriter** (p. 1113) has discovered a **DDSDataReader** (p. 1087) for the same **DDSTopic** (p. 1419) and common partition, but with a requested QoS that is incompatible with that offered by the **DDSDataWriter** (p. 1113).

Parameters:

writer *<<out>>* *(p. 200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

status *<<out>>* *(p. 200)* Current incompatible qos status of locally created **DDSDataWriter** (p. 1113)

6.182.2.4 virtual void DDSDataWriterListener::on_publication_matched (DDSDataWriter * *writer*, const DDS_PublicationMatchedStatus & *status*) [virtual]

Handles the **DDS_PUBLICATION_MATCHED_STATUS** (p. 325) status.

This callback is called when the **DDSDataWriter** (p. 1113) has found a **DDSDataReader** (p. 1087) that matches the **DDSTopic** (p. 1419), has a common partition and compatible QoS, or has ceased to be matched with a **DDSDataReader** (p. 1087) that was previously considered to be matched.

Parameters:

writer <<*out*>> *(p. 200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

status <<*out*>> *(p. 200)* Current publication match status of locally created **DDSDataWriter** (p. 1113)

6.182.2.5 virtual void DDSDataWriterListener::on_reliable_writer_cache_changed (DDSDataWriter * *writer*, const DDS_ReliableWriterCacheChangedStatus & *status*) [virtual]

<<*eXtension*>> *(p. 199)* A change has occurred in the writer's cache of unacknowledged samples.

Parameters:

writer <<*out*>> *(p. 200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

status <<*out*>> *(p. 200)* Current reliable writer cache changed status of locally created **DDSDataWriter** (p. 1113)

6.182.2.6 virtual void DDSDataWriterListener::on_reliable_reader_activity_changed (DDSDataWriter * *writer*, const DDS_ReliableReaderActivityChangedStatus & *status*) [virtual]

<<*eXtension*>> *(p. 199)* A matched reliable reader has become active or become inactive.

Parameters:

writer <<*out*>> *(p. 200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

status <<***out***>> *(*p. *200)* Current reliable reader activity changed status of locally created **DDSDataWriter** (p. 1113)

6.182.2.7 virtual void DDSDataWriterListener::on_instance_replaced (DDSDataWriter * *writer*, const DDS_InstanceHandle_t & *handle*) [virtual]

Notifies when an instance is replaced in DataWriter queue.

This callback is called when an instance is replaced by the **DDSDataWriter** (p. 1113) due to instance resource limits being reached. This callback returns to the user the handle of the replaced instance, which can be used to get the key of the replaced instance.

Parameters:

writer <<***out***>> *(*p. *200)* Locally created **DDSDataWriter** (p. 1113) that triggers the listener callback

handle <<***out***>> *(*p. *200)* Handle of the replaced instance

6.183 DDSDomainEntity Class Reference

<<*interface*>> *(p. 199)* Abstract base class for all DDS entities except for the **DDSDomainParticipant** (p. 1139).

Inheritance diagram for DDSDomainEntity::

6.183.1 Detailed Description

<<*interface*>> *(p. 199)* Abstract base class for all DDS entities except for the **DDSDomainParticipant** (p. 1139).

Its sole purpose is to *conceptually* express that **DDSDomainParticipant** (p. 1139) is a special kind of **DDSEntity** (p. 1253) that acts as a container of all other **DDSEntity** (p. 1253) but itself cannot contain other **DDSDomainParticipant** (p. 1139).

6.184 DDSDomainParticipant Class Reference

<<*interface*>> *(p. 199)* Container for all **DDSDomainEntity** (p. 1138) objects.

Inheritance diagram for DDSDomainParticipant::

Public Member Functions

- virtual **DDS_ReturnCode_t get_default_datawriter_qos** (**DDS_DataWriterQos** &qos)=0

  <<**eXtension**>> (p. 199) *Copy the default* ***DDS_DataWriterQos*** *(p. 553) values into the provided* ***DDS_DataWriterQos*** *(p. 553) instance.*

- virtual **DDS_ReturnCode_t set_default_datawriter_qos** (const **DDS_DataWriterQos** &qos)=0

  <<**eXtension**>> (p. 199) *Set the default DataWriterQos values for this DomainParticipant.*

- virtual **DDS_ReturnCode_t set_default_datawriter_qos_with_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Set the default* ***DDS_DataWriterQos*** *(p. 553) values for this domain participant based on the input XML QoS profile.*

- virtual **DDS_ReturnCode_t get_default_datareader_qos** (**DDS_DataReaderQos** &qos)=0

  <<**eXtension**>> (p. 199) *Copy the default* ***DDS_DataReaderQos*** *(p. 515) values into the provided* ***DDS_DataReaderQos*** *(p. 515) instance.*

- virtual **DDS_ReturnCode_t set_default_datareader_qos** (const **DDS_DataReaderQos** &qos)=0

  <<**eXtension**>> (p. 199) *Set the default* ***DDS_DataReaderQos*** *(p. 515) values for this domain participant.*

- virtual **DDS_ReturnCode_t set_default_datareader_qos_with_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Set the default* ***DDS_DataReaderQos*** *(p. 515) values for this DomainParticipant based on the input XML QoS profile.*

- virtual **DDS_ReturnCode_t get_default_flowcontroller_property** (**DDS_FlowControllerProperty_t** &prop)=0

<<**eXtension**>> (p. 199) *Copies the default* ***DDS_FlowControllerProperty_t*** *(p. 749) values for this domain participant into the given* ***DDS_FlowControllerProperty_t*** *(p. 749) instance.*

^ virtual **DDS_ReturnCode_t set_default_flowcontroller_property** (const **DDS_FlowControllerProperty_t** &prop)=0

<<**eXtension**>> (p. 199) *Set the default* ***DDS_FlowControllerProperty_t*** *(p. 749) values for this domain participant.*

^ virtual **DDS_ReturnCode_t register_contentfilter** (const char *filter_name, const **DDSContentFilter** *contentfilter)=0

<<**eXtension**>> (p. 199) *Register a content filter which can be used to create a* ***DDSContentFilteredTopic*** *(p. 1081).*

^ virtual **DDSContentFilter** * **lookup_contentfilter** (const char *filter_name)=0

<<**eXtension**>> (p. 199) *Lookup a content filter previously registered with* ***DDSDomainParticipant::register_contentfilter*** *(p. 1156).*

^ virtual **DDS_ReturnCode_t unregister_contentfilter** (const char *filter_name)=0

<<**eXtension**>> (p. 199) *Unregister a content filter previously registered with* ***DDSDomainParticipant::register_contentfilter*** *(p. 1156).*

^ virtual const char * **get_default_library** ()=0

<<**eXtension**>> (p. 199) *Gets the default XML library associated with a* ***DDSDomainParticipant*** *(p. 1139).*

^ virtual const char * **get_default_profile** ()=0

<<**eXtension**>> (p. 199) *Gets the default XML profile associated with a* ***DDSDomainParticipant*** *(p. 1139).*

^ virtual const char * **get_default_profile_library** ()=0

<<**eXtension**>> (p. 199) *Gets the library where the default XML QoS profile is contained for a* ***DDSDomainParticipant*** *(p. 1139).*

^ virtual **DDS_ReturnCode_t set_default_library** (const char *library_name)=0

<<**eXtension**>> (p. 199) *Sets the default XML library for a* ***DDSDomainParticipant*** *(p. 1139).*

^ virtual **DDS_ReturnCode_t set_default_profile** (const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Sets the default XML profile for a* ***DDSDomainParticipant*** *(p. 1139).*

^ virtual **DDS_ReturnCode_t get_default_topic_qos** (**DDS_TopicQos** &qos)=0

*Copies the default **DDS_TopicQos** (p. 965) values for this domain participant into the given **DDS_TopicQos** (p. 965) instance.*

^ virtual **DDS_ReturnCode_t set_default_topic_qos** (const **DDS_TopicQos** &qos)=0

*Set the default **DDS_TopicQos** (p. 965) values for this domain participant.*

^ virtual **DDS_ReturnCode_t set_default_topic_qos_with_profile** (const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Set the default **DDS_TopicQos** (p. 965) values for this domain participant based on the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t get_default_publisher_qos** (**DDS_PublisherQos** &qos)=0

*Copy the default **DDS_PublisherQos** (p. 851) values into the provided **DDS_PublisherQos** (p. 851) instance.*

^ virtual **DDS_ReturnCode_t set_default_publisher_qos** (const **DDS_PublisherQos** &qos)=0

*Set the default **DDS_PublisherQos** (p. 851) values for this DomainParticipant.*

^ virtual **DDS_ReturnCode_t set_default_publisher_qos_with_profile** (const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Set the default **DDS_PublisherQos** (p. 851) values for this DomainParticipant based on the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t get_default_subscriber_qos** (**DDS_SubscriberQos** &qos)=0

*Copy the default **DDS_SubscriberQos** (p. 934) values into the provided **DDS_SubscriberQos** (p. 934) instance.*

^ virtual **DDS_ReturnCode_t set_default_subscriber_qos** (const **DDS_SubscriberQos** &qos)=0

*Set the default **DDS_SubscriberQos** (p. 934) values for this Domain{articipant.*

^ virtual **DDS_ReturnCode_t set_default_subscriber_qos_with_profile** (const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Set the default **DDS_SubscriberQos** (p. 934) values for this DomainParticipant based on the input XML QoS profile.*

^ virtual **DDSPublisher** * **create_publisher** (const **DDS_PublisherQos** &qos, **DDSPublisherListener** *listener, **DDS_StatusMask** mask)=0

*Creates a **DDSPublisher** (p. 1346) with the desired QoS policies and attaches to it the specified **DDSPublisherListener** (p. 1370).*

^ virtual **DDSPublisher** * **create_publisher_with_profile** (const char *library_name, const char *profile_name, **DDSPublisherListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a new **DDSPublisher** (p. 1346) object using the **DDS_PublisherQos** (p. 851) associated with the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t** **delete_publisher** (**DDSPublisher** *p)=0

*Deletes an existing **DDSPublisher** (p. 1346).*

^ virtual **DDSSubscriber** * **create_subscriber** (const **DDS_SubscriberQos** &qos, **DDSSubscriberListener** *listener, **DDS_StatusMask** mask)=0

*Creates a **DDSSubscriber** (p. 1390) with the desired QoS policies and attaches to it the specified **DDSSubscriberListener** (p. 1414).*

^ virtual **DDSSubscriber** * **create_subscriber_with_profile** (const char *library_name, const char *profile_name, **DDSSubscriberListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a new **DDSSubscriber** (p. 1390) object using the **DDS_PublisherQos** (p. 851) associated with the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t** **delete_subscriber** (**DDSSubscriber** *s)=0

*Deletes an existing **DDSSubscriber** (p. 1390).*

^ virtual **DDS_ReturnCode_t** **get_publishers** (**DDSPublisherSeq** &publishers)=0

<<**eXtension**>> (p. 199) *Allows the application to access all the publishers the participant has.*

^ virtual **DDS_ReturnCode_t** **get_subscribers** (**DDSSubscriberSeq** &subscribers)=0

<<**eXtension**>> (p. 199) *Allows the application to access all the subscribers the participant has.*

^ virtual **DDSTopic** * **create_topic** (const char *topic_name, const char *type_name, const **DDS_TopicQos** &qos, **DDSTopicListener** *listener, **DDS_StatusMask** mask)=0

*Creates a **DDSTopic** (p. 1419) with the desired QoS policies and attaches to it the specified **DDSTopicListener** (p. 1430).*

^ virtual **DDSTopic** * **create_topic_with_profile** (const char *topic_name, const char *type_name, const char *library_name, const char *profile_name, **DDSTopicListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a new **DDSTopic** (p. 1419) object using the **DDS_PublisherQos** (p. 851) associated with the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t delete_topic** (**DDSTopic** *topic)=0

*Deletes a **DDSTopic** (p. 1419).*

^ virtual **DDSContentFilteredTopic** * **create_contentfilteredtopic** (const char *name, **DDSTopic** *related_topic, const char *filter_expression, const **DDS_StringSeq** &expression_parameters)=0

*Creates a **DDSContentFilteredTopic** (p. 1081), that can be used to do content-based subscriptions.*

^ virtual **DDSContentFilteredTopic** * **create_contentfilteredtopic_with_filter** (const char *name, **DDSTopic** *related_topic, const char *filter_expression, const **DDS_StringSeq** &expression_parameters, const char *filter_name=**DDS_SQLFILTER_NAME**)=0

<<**eXtension**>> (p. 199) *Creates a **DDSContentFilteredTopic** (p. 1081) using the specified filter to do content-based subscriptions.*

^ virtual **DDS_ReturnCode_t delete_contentfilteredtopic** (**DDSContentFilteredTopic** *a_contentfilteredtopic)=0

*Deletes a **DDSContentFilteredTopic** (p. 1081).*

^ virtual **DDSMultiTopic** * **create_multitopic** (const char *name, const char *type_name, const char *subscription_expression, const **DDS_StringSeq** &expression_parameters)=0

*[**Not supported (optional)**] Creates a MultiTopic that can be used to subscribe to multiple topics and combine/filter the received data into a resulting type.*

^ virtual **DDS_ReturnCode_t delete_multitopic** (**DDSMultiTopic** *a_multitopic)=0

*[**Not supported (optional)**] Deletes a **DDSMultiTopic** (p. 1322).*

^ virtual **DDSTopic** * **find_topic** (const char *topic_name, const **DDS_Duration_t** &timeout)=0

*Finds an existing (or ready to exist) **DDSTopic** (p. 1419), based on its name.*

^ virtual **DDSTopicDescription** * **lookup_topicdescription** (const char *topic_name)=0

*Looks up an existing, locally created **DDSTopicDescription** (p. 1427), based on its name.*

^ virtual **DDSFlowController** * **create_flowcontroller** (const char *name, const **DDS_FlowControllerProperty_t** &prop)=0

<<**eXtension**>> (p. 199) *Creates a **DDSFlowController** (p. 1259) with the desired property.*

^ virtual **DDS_ReturnCode_t** **delete_flowcontroller** (**DDSFlowController** *fc)=0

<<**eXtension**>> (p. 199) *Deletes an existing **DDSFlowController** (p. 1259).*

^ virtual **DDSFlowController** * **lookup_flowcontroller** (const char *name)=0

<<**eXtension**>> (p. 199) *Looks up an existing locally-created **DDSFlowController** (p. 1259), based on its name.*

^ virtual **DDSSubscriber** * **get_builtin_subscriber** ()=0

Accesses the **built-in** ***DDSSubscriber*** *(p. 1390).*

^ virtual **DDS_ReturnCode_t** **ignore_participant** (const **DDS_InstanceHandle_t** &handle)=0

*Instructs RTI Connext to locally ignore a remote **DDSDomainParticipant** (p. 1139).*

^ virtual **DDS_ReturnCode_t** **ignore_topic** (const **DDS_InstanceHandle_t** &handle)=0

*Instructs RTI Connext to locally ignore a **DDSTopic** (p. 1419).*

^ virtual **DDS_ReturnCode_t** **ignore_publication** (const **DDS_InstanceHandle_t** &handle)=0

Instructs RTI Connext to locally ignore a publication.

- virtual **DDS_ReturnCode_t ignore_subscription** (const **DDS_InstanceHandle_t** &handle)=0

  Instructs RTI Connext to locally ignore a subscription.

- virtual **DDS_DomainId_t get_domain_id** ()=0

  Get the unique domain identifier.

- virtual **DDS_ReturnCode_t get_current_time** (**DDS_Time_t** &current_time)=0

  Returns the current value of the time.

- virtual **DDS_ReturnCode_t register_durable_subscription** (**DDS_EndpointGroup_t** &group, const char *topic_name)=0

  *Registers a Durable Subscription on the specified **DDSTopic** (p. 1419) on all Persistence Services.*

- virtual **DDS_ReturnCode_t delete_durable_subscription** (**DDS_EndpointGroup_t** &group)=0

  Deletes an existing Durable Subscription on all Persistence Services.

- virtual **DDS_ReturnCode_t assert_liveliness** ()=0

  *Manually asserts the liveliness of this **DDSDomainParticipant** (p. 1139).*

- virtual **DDS_ReturnCode_t delete_contained_entities** ()=0

  *Delete all the entities that were created by means of the "create" operations on the **DDSDomainParticipant** (p. 1139).*

- virtual **DDS_ReturnCode_t get_discovered_participants** (**DDS_InstanceHandleSeq** &participant_handles)=0

  *Returns list of discovered **DDSDomainParticipant** (p. 1139) s.*

- virtual **DDS_ReturnCode_t get_discovered_participant_data** (struct **DDS_ParticipantBuiltinTopicData** &participant_data, const **DDS_InstanceHandle_t** &participant_handle)=0

  *Returns **DDS_ParticipantBuiltinTopicData** (p. 816) for the specified **DDSDomainParticipant** (p. 1139) .*

- virtual **DDS_ReturnCode_t get_discovered_topics** (**DDS_InstanceHandleSeq** &topic_handles)=0

  *Returns list of discovered **DDSTopic** (p. 1419) objects.*

- virtual **DDS_ReturnCode_t get_discovered_topic_data** (struct **DDS_TopicBuiltinTopicData** &topic_data, const **DDS_InstanceHandle_t** &topic_handle)=0

Returns ***DDS_TopicBuiltinTopicData*** *(p. 958) for the specified* ***DDSTopic*** *(p. 1419).*

^ virtual **DDS_Boolean contains_entity** (const **DDS_InstanceHandle_t** &a_handle)=0

Completes successfully with ***DDS_BOOLEAN_TRUE*** *(p. 298) if the referenced* ***DDSEntity*** *(p. 1253) is contained by the* ***DDSDomainParticipant*** *(p. 1139).*

^ virtual **DDS_ReturnCode_t set_qos** (const **DDS_DomainParticipantQos** &qos)=0

Change the QoS of this DomainParticipant.

^ virtual **DDS_ReturnCode_t set_qos_with_profile** (const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Change the QoS of this domain participant using the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t get_qos** (**DDS_DomainParticipantQos** &qos)=0

Get the participant QoS.

^ virtual **DDS_ReturnCode_t add_peer** (const char *peer_desc_string)=0

<<**eXtension**>> (p. 199) *Attempt to contact one or more additional peer participants.*

^ virtual **DDS_ReturnCode_t remove_peer** (const char *peer_desc_string)=0

<<**eXtension**>> (p. 199) *Remove one or more peer participants from the list of peers with which this* ***DDSDomainParticipant*** *(p. 1139) will try to communicate.*

^ virtual **DDS_ReturnCode_t set_listener** (**DDSDomainParticipantListener** *l, **DDS_StatusMask** mask=DDS_STATUS_MASK_ALL)=0

Sets the participant listener.

^ virtual **DDSDomainParticipantListener** * **get_listener** ()=0

Get the participant listener.

^ virtual **DDSPublisher** * **get_implicit_publisher** ()=0

<<**eXtension**>> (p. 199) *Returns the implicit* ***DDSPublisher*** *(p. 1346). If an implicit Publisher does not already exist, this creates one.*

^ virtual **DDSSubscriber** * **get_implicit_subscriber** ()=0

<<**eXtension**>> (p. 199) *Returns the implicit **DDSSubscriber** (p. 1390). If an implicit Subscriber does not already exist, this creates one.*

^ virtual **DDSDataWriter** * **create_datawriter** (**DDSTopic** *topic, const **DDS_DataWriterQos** &qos, **DDSDataWriterListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a **DDSDataWriter** (p. 1113) that will be attached and belong to the implicit **DDSPublisher** (p. 1346).*

^ virtual **DDSDataWriter** * **create_datawriter_with_profile** (**DDSTopic** *topic, const char *library_name, const char *profile_name, **DDSDataWriterListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a **DDSDataWriter** (p. 1113) using a XML QoS profile that will be attached and belong to the implicit **DDSPublisher** (p. 1346).*

^ virtual **DDS_ReturnCode_t** **delete_datawriter** (**DDSDataWriter** *a_datawriter)=0

<<**eXtension**>> (p. 199) *Deletes a **DDSDataWriter** (p. 1113) that belongs to the implicit **DDSPublisher** (p. 1346).*

^ virtual **DDSDataReader** * **create_datareader** (**DDSTopicDescription** *topic, const **DDS_DataReaderQos** &qos, **DDSDataReaderListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a **DDSDataReader** (p. 1087) that will be attached and belong to the implicit **DDSSubscriber** (p. 1390).*

^ virtual **DDSDataReader** * **create_datareader_with_profile** (**DDSTopicDescription** *topic, const char *library_name, const char *profile_name, **DDSDataReaderListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a **DDSDataReader** (p. 1087) using a XML QoS profile that will be attached and belong to the implicit **DDSSubscriber** (p. 1390).*

^ virtual **DDS_ReturnCode_t** **delete_datareader** (**DDSDataReader** *a_datareader)=0

<<**eXtension**>> (p. 199) *Deletes a **DDSDataReader** (p. 1087) that belongs to the implicit **DDSSubscriber** (p. 1390).*

^ virtual **DDSPublisher** * **lookup_publisher_by_name_exp** (const char *publisher_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Looks up a **DDSPublisher** (p. 1346) by its entity name within this **DDSDomainParticipant** (p. 1139).*

ˆ virtual **DDSSubscriber** ∗ **lookup_subscriber_by_name_exp** (const char ∗subscriber_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Retrieves a **DDSSubscriber** (p. 1390) by its entity name within this **DDSDomainParticipant** (p. 1139).*

ˆ virtual **DDSDataWriter** ∗ **lookup_datawriter_by_name_exp** (const char ∗datawriter_full_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Looks up a **DDSDataWriter** (p. 1113) by its entity name within this **DDSDomainParticipant** (p. 1139).*

ˆ virtual **DDSDataReader** ∗ **lookup_datareader_by_name_exp** (const char ∗datareader_full_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Retrieves up a **DDSDataReader** (p. 1087) by its entity name in this **DDSDomainParticipant** (p. 1139).*

6.184.1 Detailed Description

<<***interface***>> *(p. 199)* Container for all **DDSDomainEntity** (p. 1138) objects.

The DomainParticipant object plays several roles:

- It acts as a container for all other **DDSEntity** (p. 1253) objects.

- It acts as *factory* for the **DDSPublisher** (p. 1346), **DDSSubscriber** (p. 1390), **DDSTopic** (p. 1419) and **DDSMultiTopic** (p. 1322) **DDSEntity** (p. 1253) objects.

- It represents the participation of the application on a communication plane that isolates applications running on the same set of physical computers from each other. A domain establishes a virtual network linking all applications that share the same `domainId` and isolating them from applications running on different domains. In this way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.

- It provides administration services in the domain, offering operations that allow the application to ignore locally any information about a given participant (**ignore_participant()** (p. 1187)), publication (**ignore_publication()**

(p. 1189)), subscription (**ignore_subscription()** (p. 1190)) or topic (**ignore_topic()** (p. 1188)).

The following operations may be called even if the **DDSDomainParticipant** (p. 1139) is not enabled. (Operations NOT in this list will fail with the value **DDS_RETCODE_NOT_ENABLED** (p. 315) if called on a disabled DomainParticipant).

- Operations defined at the base-class level: **set_qos()** (p. 1197), **set_qos_with_profile()** (p. 1198), **get_qos()** (p. 1199), **set_listener()** (p. 1202), **get_listener()** (p. 1202), **enable()** (p. 1256);

- Factory operations: **create_flowcontroller()** (p. 1184), **create_topic()** (p. 1175), **create_topic_with_profile()** (p. 1177), **create_publisher()** (p. 1169), **create_publisher_with_profile()** (p. 1170), **create_subscriber()** (p. 1172), **create_subscriber_with_profile()** (p. 1173), **delete_flowcontroller()** (p. 1185), **delete_topic()** (p. 1178), **delete_publisher()** (p. 1171), **delete_subscriber()** (p. 1173), **set_default_flowcontroller_property()** (p. 1155), **get_default_flowcontroller_property()** (p. 1154), **set_default_topic_qos()** (p. 1162), **set_default_topic_qos_with_profile()** (p. 1162), **get_default_topic_qos()** (p. 1161), **set_default_publisher_qos()** (p. 1164), **set_default_publisher_qos_with_profile()** (p. 1165), **get_default_publisher_qos()** (p. 1163), **set_default_subscriber_qos()** (p. 1167), **set_default_subscriber_qos_with_profile()** (p. 1168), **get_default_subscriber_qos()** (p. 1166), **delete_contained_entities()** (p. 1193), **set_default_datareader_qos()** (p. 1153), **set_default_datareader_qos_with_profile()** (p. 1154), **get_default_datareader_qos()** (p. 1152), **set_default_datawriter_qos()** (p. 1150), **set_default_datawriter_qos_with_profile()** (p. 1151), **get_default_datawriter_qos()** (p. 1150), **set_default_library()** (p. 1159), **set_default_profile()** (p. 1160);

- Operations for looking up topics: **lookup_topicdescription()** (p. 1183);

- Operations that access status: **get_statuscondition()** (p. 1257), **get_status_changes()** (p. 1257).

QoS:

DDS_DomainParticipantQos (p. 588)

Status:

Status Kinds (p. 317)

Listener:

DDSDomainParticipantListener (p. 1243)

See also:

Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

6.184.2 Member Function Documentation

6.184.2.1 virtual DDS_ReturnCode_t DDSDomainParticipant::get_default_datawriter_qos (DDS_DataWriterQos & *qos*) [pure virtual]

<<eXtension>> (p. 199) Copy the default **DDS_DataWriterQos** (p. 553) values into the provided **DDS_DataWriterQos** (p. 553) instance.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSDomainParticipant::set_default_datawriter_qos** (p. 1150), or **DDSDomainParticipant::set_default_datawriter_qos_with_profile** (p. 1151), or else, if the call was never made, the default values listed in **DDS_DataWriterQos** (p. 553).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

MT Safety:

UNSAFE. It is not safe to retrieve the default DataWriter QoS from a DomainPartipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datawriter_qos** (p. 1150).

Parameters:

qos *<<inout>> (p. 200)* Qos to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

6.184.2.2 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_datawriter_qos (const DDS_DataWriterQos & *qos*) [pure virtual]

<<eXtension>> (p. 199) Set the default DataWriterQos values for this DomainParticipant.

This set of default values will be inherited for a newly created **DDSPublisher** (p. 1346).

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default DataWriter QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datawriter_qos** (p. 1150) or **DDSDomainParticipant::get_default_datawriter_qos** (p. 1150).

Parameters:

qos *<<in>>* *(p. 200)* Default qos to be set. The special value **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSDomainParticipant::set_default_datawriter_qos** (p. 1150) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.184.2.3 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_datawriter_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_DataWriterQos** (p. 553) values for this domain participant based on the input XML QoS profile.

This set of default values will be inherited for a newly created **DDSPublisher** (p. 1346).

Precondition:

The **DDS_DataWriterQos** (p. 553) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default DataWriter QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datawriter_qos** (p. 1150) or **DDSDomainParticipant::get_default_datawriter_qos** (p. 1150)

Parameters:

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.184.2.4 virtual DDS_ReturnCode_t DDSDomainParticipant::get_default_datareader_qos (DDS_DataReaderQos & *qos*) [pure virtual]

<<eXtension>> *(p. 199)* Copy the default **DDS_DataReaderQos** (p. 515) values into the provided **DDS_DataReaderQos** (p. 515) instance.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSDomainParticipant::set_default_datareader_qos** (p. 1153), or **DDSDomainParticipant::set_default_datareader_qos_with_profile** (p. 1154), or else, if the call was never made, the default values listed in **DDS_DataReaderQos** (p. 515).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

MT Safety:

UNSAFE. It is not safe to retrieve the default DataReader QoS from a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datareader_qos** (p. 1153).

Parameters:

qos *<<**inout**>>* *(p. 200)* Qos to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

6.184.2.5 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_datareader_qos (const DDS_DataReaderQos & *qos*) [pure virtual]

*<<**eXtension**>>* *(p. 199)* Set the default **DDS_DataReaderQos** (p. 515) values for this domain participant.

This set of default values will be inherited for a newly created **DDSSubscriber** (p. 1390).

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default DataReader QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datareader_qos** (p. 1153) or **DDSDomainParticipant::get_default_datareader_qos** (p. 1152).

Parameters:

qos *<<**in**>>* *(p. 200)* Default qos to be set. The special value **DDS_DATAREADER_QOS_DEFAULT** (p. 99) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSDomainParticipant::set_default_datareader_qos** (p. 1153) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.184.2.6 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_datareader_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_DataReaderQos** (p. 515) values for this DomainParticipant based on the input XML QoS profile.

This set of default values will be inherited for a newly created **DDSSubscriber** (p. 1390).

Precondition:

The **DDS_DataReaderQos** (p. 515) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default DataReader QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datareader_qos** (p. 1153) or **DDSDomainParticipant::get_default_datareader_qos** (p. 1152).

Parameters:

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.184.2.7 virtual DDS_ReturnCode_t DDSDomainParticipant::get_default_flowcontroller_property (DDS_FlowControllerProperty_t & *prop*) [pure virtual]

<<eXtension>> *(p. 199)* Copies the default **DDS_FlowControllerProperty_t** (p. 749) values for this domain participant into the given **DDS_FlowControllerProperty_t** (p. 749) instance.

The retrieved `property` will match the set of values specified on the last successful call to **DDSDomainParticipant::set_default_flowcontroller_property** (p. 1155), or else, if the call was never made, the default values listed in **DDS_FlowControllerProperty_t** (p. 749).

MT Safety:

UNSAFE. It is not safe to retrieve the default flow controller properties from a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_flowcontroller_property** (p. 1155)

Parameters:

prop *<<in>>* *(p. 200)* Default property to be retrieved.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT (p. 40)
DDSDomainParticipant::create_flowcontroller (p. 1184)

6.184.2.8 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_flowcontroller_property (const DDS_FlowControllerProperty_t & *prop*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_FlowControllerProperty_t** (p. 749) values for this domain participant.

This default value will be used for newly created **DDSFlowController** (p. 1259) if **DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT** (p. 40) is specified as the `property` parameter when **DDSDomainParticipant::create_flowcontroller** (p. 1184) is called.

Precondition:

The specified property values must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default flow controller properties for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_flowcontroller_property**

(p. 1155) , **DDSDomainParticipant::get_default_flowcontroller_property** (p. 1154) or calling **DDSDomainParticipant::create_flowcontroller** (p. 1184) with **DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT** (p. 40) as the `qos` parameter.

Parameters:

prop *<<in>> (p. 200)* Default property to be set. The special value **DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT** (p. 40) may be passed as `property` to indicate that the default property should be reset to the default values the factory would use if **DDSDomainParticipant::set_default_flowcontroller_property** (p. 1155) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT (p. 40)
DDSDomainParticipant::create_flowcontroller (p. 1184)

6.184.2.9 virtual DDS_ReturnCode_t DDSDomainParticipant::register_contentfilter (const char * *filter_name*, const DDSContentFilter * *contentfilter*) [pure virtual]

<<eXtension>> (p. 199) Register a content filter which can be used to create a **DDSContentFilteredTopic** (p. 1081).

DDS specifies a SQL-like content filter for use by content filtered topics. If this filter does not meet your filtering requirements, you can register a custom filter.

To use a custom filter, it must be registered in the following places:

- In any application that uses the custom filter to create a **DDSContentFilteredTopic** (p. 1081) and the corresponding **DDSDataReader** (p. 1087).
- In each application that writes the data to the applications mentioned above.

For example, suppose Application A on the subscription side creates a Topic named X and a ContentFilteredTopic named filteredX (and a corresponding DataReader), using a previously registered content filter, myFilter. With only that, you will have filtering at the subscription side. If you also want to perform

filtering in any application that publishes Topic X, then you also need to register the same definition of the ContentFilter myFilter in that application.

Each `filter_name` can only be used to registered a content filter once with a **DDSDomainParticipant** (p. 1139).

Parameters:

filter_name *<<in>>* *(p. 200)* Name of the filter. The name must be unique within the **DDSDomainParticipant** (p. 1139) and must not exceed 255 characters. Cannot be NULL.

contentfilter *<<in>>* *(p. 200)* Content filter to be registered. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDomainParticipant::unregister_contentfilter (p. 1158)

6.184.2.10 virtual DDSContentFilter* DDSDomainParticipant::lookup_contentfilter (const char * *filter_name*) [pure virtual]

<<eXtension>> *(p. 199)* Lookup a content filter previously registered with **DDSDomainParticipant::register_contentfilter** (p. 1156).

Parameters:

filter_name *<<in>>* *(p. 200)* Name of the filter. Cannot be NULL.

Returns:

NULL if the given filter_name has not been previously registered to the **DDSDomainParticipant** (p. 1139) with **DDSDomainParticipant::register_contentfilter** (p. 1156). Otherwise, return the **DDSContentFilter** (p. 1077) that has been previosuly registered with the given filter_name.

See also:

DDSDomainParticipant::register_contentfilter (p. 1156)

6.184.2.11 virtual DDS_ReturnCode_t DDSDomainParticipant::unregister_contentfilter (const char * *filter_name*) [pure virtual]

<<eXtension>> (p. 199) Unregister a content filter previously registered with **DDSDomainParticipant::register_contentfilter** (p. 1156).

A `filter_name` can be unregistered only if it has been previously registered to the **DDSDomainParticipant** (p. 1139) with **DDSDomainParticipant::register_contentfilter** (p. 1156).

The unregistration of filter is not allowed if there are any existing **DDSContentFilteredTopic** (p. 1081) objects that are using the filter. If the operation is called on a filter with existing **DDSContentFilteredTopic** (p. 1081) objects attached to it, this operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

If there are still existing discovered **DDSDataReader** (p. 1087) s with the same `filter_name` and the filter's compile method of the filter have previously been called on the discovered **DDSDataReader** (p. 1087) s, finalize method of the filter will be called on those discovered **DDSDataReader** (p. 1087) s before the content filter is unregistered. This means filtering will now be performed on the application that is creating the **DDSDataReader** (p. 1087).

Parameters:

filter_name *<<in>> (p. 200)* Name of the filter. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

See also:

DDSDomainParticipant::register_contentfilter (p. 1156)

6.184.2.12 virtual const char* DDSDomainParticipant::get_default_library () [pure virtual]

<<eXtension>> (p. 199) Gets the default XML library associated with a **DDSDomainParticipant** (p. 1139).

Returns:

The default library or null if the default library was not set.

See also:

DDSDomainParticipant::set_default_library (p. 1159)

6.184.2.13 virtual const char* DDSDomainParticipant::get_default_profile () [pure virtual]

<<*eXtension*>> *(p. 199)* Gets the default XML profile associated with a **DDSDomainParticipant** (p. 1139).

Returns:

The default profile or null if the default profile was not set.

See also:

DDSDomainParticipant::set_default_profile (p. 1160)

6.184.2.14 virtual const char* DDSDomainParticipant::get_default_profile_library () [pure virtual]

<<*eXtension*>> *(p. 199)* Gets the library where the default XML QoS profile is contained for a **DDSDomainParticipant** (p. 1139).

The default profile library is automatically set when **DDSDomainParticipant::set_default_profile** (p. 1160) is called.

This library can be different than the **DDSDomainParticipant** (p. 1139) default library (see **DDSDomainParticipant::get_default_library** (p. 1158)).

Returns:

The default profile library or null if the default profile was not set.

See also:

DDSDomainParticipant::set_default_profile (p. 1160)

6.184.2.15 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_library (const char * *library_name*) [pure virtual]

<<*eXtension*>> *(p. 199)* Sets the default XML library for a **DDSDomainParticipant** (p. 1139).

This method specifies the library that will be used as the default the next time a default library is needed during a call to one of this DomainParticipant's operations.

Any API requiring a library_name as a parameter can use null to refer to the default library.

If the default library is not set, the **DDSDomainParticipant** (p. 1139) inherits the default from the **DDSDomainParticipantFactory** (p. 1216) (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

Parameters:

library_name *<<in>>* *(p. 200)* Library name. If library_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDomainParticipant::get_default_library (p. 1158)

6.184.2.16 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Sets the default XML profile for a **DDSDomainParticipant** (p. 1139).

This method specifies the profile that will be used as the default the next time a default DomainParticipant profile is needed during a call to one of this DomainParticipant's operations. When calling a **DDSDomainParticipant** (p. 1139) method that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

If the default profile is not set, the **DDSDomainParticipant** (p. 1139) inherits the default from the **DDSDomainParticipantFactory** (p. 1216) (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

This method does not set the default QoS for entities created by the **DDSDomainParticipant** (p. 1139); for this functionality, use the methods set_default_<entity>_qos_with_profile (you may pass in NULL after having called **set_default_profile()** (p. 1160)).

This method does not set the default QoS for newly created DomainParticipants; for this functionality, use **DDSDomainParticipantFactory::set_default_participant_qos_with_profile** (p. 1223).

Parameters:

library_name *<<in>>* *(p. 200)* The library name containing the profile.

profile_name *<<in>>* *(p. 200)* The profile name. If profile_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDomainParticipant::get_default_profile (p. 1159)
DDSDomainParticipant::get_default_profile_library (p. 1159)

6.184.2.17 virtual DDS_ReturnCode_t DDSDomainParticipant::get_default_topic_qos (DDS_TopicQos & *qos*) [pure virtual]

Copies the default **DDS_TopicQos** (p. 965) values for this domain participant into the given **DDS_TopicQos** (p. 965) instance.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSDomainParticipant::set_default_topic_qos** (p. 1162), or else, if the call was never made, the default values listed in **DDS_TopicQos** (p. 965).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

MT Safety:

UNSAFE. It is not safe to retrieve the default Topic QoS from a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_topic_qos** (p. 1162)

Parameters:

qos *<<in>>* *(p. 200)* Default qos to be retrieved.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_TOPIC_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_topic (p. 1175)

6.184.2.18 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_topic_qos (const DDS_TopicQos & *qos*) [pure virtual]

Set the default **DDS_TopicQos** (p. 965) values for this domain participant.

This default value will be used for newly created **DDSTopic** (p. 1419) if **DDS_TOPIC_QOS_DEFAULT** (p. 38) is specified as the `qos` parameter when **DDSDomainParticipant::create_topic** (p. 1175) is called.

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default topic QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_topic_qos** (p. 1162), **DDSDomainParticipant::get_default_topic_qos** (p. 1161) or calling **DDSDomainParticipant::create_topic** (p. 1175) with **DDS_TOPIC_QOS_DEFAULT** (p. 38) as the `qos` parameter.

Parameters:

qos *<<in>>* *(p. 200)* Default qos to be set. The special value **DDS_TOPIC_QOS_DEFAULT** (p. 38) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSDomainParticipant::set_default_topic_qos** (p. 1162) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_TOPIC_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_topic (p. 1175)

6.184.2.19 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_topic_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_TopicQos** (p. 965) values for

this domain participant based on the input XML QoS profile.

This default value will be used for newly created **DDSTopic** (p. 1419) if **DDS_TOPIC_QOS_DEFAULT** (p. 38) is specified as the `qos` parameter when **DDSDomainParticipant::create_topic** (p. 1175) is called.

Precondition:

The **DDS_TopicQos** (p. 965) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default topic QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_topic_qos** (p. 1162), **DDSDomainParticipant::get_default_topic_qos** (p. 1161) or calling **DDSDomainParticipant::create_topic** (p. 1175) with **DDS_TOPIC_QOS_DEFAULT** (p. 38) as the `qos` parameter.

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

If the input profile cannot be found the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_TOPIC_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_topic_with_profile (p. 1177)

6.184.2.20 virtual DDS_ReturnCode_t DDSDomainParticipant::get_default_publisher_qos (DDS_PublisherQos & *qos*) [pure virtual]

Copy the default **DDS_PublisherQos** (p. 851) values into the provided **DDS_PublisherQos** (p. 851) instance.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSDomainParticipant::set_default_publisher_qos** (p. 1164), or **DDSDomainParticipant::set_default_publisher_qos_with_profile** (p. 1165), or else, if the call was never made, the default values listed in **DDS_PublisherQos** (p. 851).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

If **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) is specified as the `qos` parameter when **DDSDomainParticipant::create_topic** (p. 1175) is called, the default value of the QoS set in the factory, equivalent to the value obtained by calling **DDSDomainParticipant::get_default_publisher_qos** (p. 1163), will be used to create the **DDSPublisher** (p. 1346).

MT Safety:

UNSAFE. It is not safe to retrieve the default publisher QoS from a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_publisher_qos** (p. 1164)

Parameters:

qos *<<**inout**>> (p. 200)* Qos to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_PUBLISHER_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_publisher (p. 1169)

6.184.2.21 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_publisher_qos (const DDS_PublisherQos & *qos*) [pure virtual]

Set the default **DDS_PublisherQos** (p. 851) values for this DomainParticipant.

This set of default values will be used for a newly created **DDSPublisher** (p. 1346) if **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) is specified as the `qos` parameter when **DDSDomainParticipant::create_publisher** (p. 1169) is called.

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default publisher QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_publisher_qos** (p. 1164), **DDSDomainParticipant::get_default_publisher_qos** (p. 1163) or calling **DDSDomainParticipant::create_publisher** (p. 1169) with **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) as the `qos` parameter.

Parameters:

qos *<<in>>* *(p. 200)* Default qos to be set. The special value **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSDomainParticipant::set_default_publisher_qos** (p. 1164) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_PUBLISHER_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_publisher (p. 1169)

6.184.2.22 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_publisher_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_PublisherQos** (p. 851) values for this DomainParticipant based on the input XML QoS profile.

This set of default values will be used for a newly created **DDSPublisher** (p. 1346) if **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) is specified as the `qos` parameter when **DDSDomainParticipant::create_publisher** (p. 1169) is called.

Precondition:

The **DDS_PublisherQos** (p. 851) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default publisher QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_publisher_qos** (p. 1164), **DDSDomainParticipant::get_default_publisher_qos** (p. 1163) or calling **DDSDomainParticipant::create_publisher** (p. 1169) with **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) as the `qos` parameter.

Parameters:

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_PUBLISHER_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_publisher_with_profile (p. 1170)

6.184.2.23 virtual DDS_ReturnCode_t DDSDomainParticipant::get_default_subscriber_qos (DDS_SubscriberQos & *qos*) [pure virtual]

Copy the default **DDS_SubscriberQos** (p. 934) values into the provided **DDS_SubscriberQos** (p. 934) instance.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167), or **DDSDomainParticipant::set_default_subscriber_qos_with_profile** (p. 1168), or else, if the call was never made, the default values listed in **DDS_SubscriberQos** (p. 934).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

If **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) is specified as the `qos` parameter when **DDSDomainParticipant::create_subscriber** (p. 1172) is called, the default value of the QoS set in the factory, equivalent to the value obtained by calling **DDSDomainParticipant::get_default_subscriber_qos** (p. 1166), will be used to create the **DDSSubscriber** (p. 1390).

MT Safety:

UNSAFE. It is not safe to retrieve the default Subscriber QoS from a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167).

Parameters:

qos *<<inout>>* *(p. 200)* Qos to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_SUBSCRIBER_QOS_DEFAULT (p. 39)
DDSDomainParticipant::create_subscriber (p. 1172)

6.184.2.24 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_subscriber_qos (const DDS_SubscriberQos & *qos*) [pure virtual]

Set the default **DDS_SubscriberQos** (p. 934) values for this Domain{articipant.

This set of default values will be used for a newly created **DDSSubscriber** (p. 1390) if **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) is specified as the `qos` parameter when **DDSDomainParticipant::create_subscriber** (p. 1172) is called.

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default Subscriber QoS for a DomainParticipant while another thread may be simultaneously calling

DDSDomainParticipant::set_default_subscriber_qos (p. 1167), **DDSDomainParticipant::get_default_subscriber_qos** (p. 1166) or calling **DDSDomainParticipant::create_subscriber** (p. 1172) with **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) as the `qos` parameter.

Parameters:

qos *<<in>>* *(p. 200)* Default qos to be set. The special value **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.184.2.25 virtual DDS_ReturnCode_t DDSDomainParticipant::set_default_subscriber_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_SubscriberQos** (p. 934) values for this DomainParticipant based on the input XML QoS profile.

This set of default values will be used for a newly created **DDSSubscriber** (p. 1390) if **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) is specified as the `qos` parameter when **DDSDomainParticipant::create_subscriber** (p. 1172) is called.

Precondition:

The **DDS_SubscriberQos** (p. 934) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default Subscriber QoS for a DomainParticipant while another thread may be simultaneously calling **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167), **DDSDomainParticipant::get_default_subscriber_qos** (p. 1166) or calling **DDSDomainParticipant::create_subscriber** (p. 1172) with **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) as the `qos` parameter.

Parameters:

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_SUBSCRIBER_QOS_DEFAULT (p. 39)
DDSDomainParticipant::create_subscriber_with_profile (p. 1173)

6.184.2.26 virtual DDSPublisher* DDSDomainParticipant::create_publisher (const DDS_PublisherQos & *qos*, DDSPublisherListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

Creates a **DDSPublisher** (p. 1346) with the desired QoS policies and attaches to it the specified **DDSPublisherListener** (p. 1370).

Precondition:

The specified QoS policies must be consistent, or the operation will fail and no **DDSPublisher** (p. 1346) will be created.

MT Safety:

UNSAFE. If **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) is used for `qos`, it is not safe to create the publisher while another thread may be simultaneously calling **DDSDomainParticipant::set_default_publisher_qos** (p. 1164).

Parameters:

qos *<<in>>* *(p. 200)* QoS to be used for creating the new **DDSPublisher** (p. 1346). The special value **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) can be used to indicate that the **DDSPublisher**

(p. 1346) should be created with the default **DDS_PublisherQos** (p. 851) set in the **DDSDomainParticipant** (p. 1139).

listener *<<in>> (p. 200).* Listener to be attached to the newly created **DDSPublisher** (p. 1346).

mask *<<in>> (p. 200).* Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

newly created publisher object or NULL on failure.

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_PublisherQos (p. 851) for rules on consistency among QoS
DDS_PUBLISHER_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_publisher_with_profile (p. 1170)
DDSDomainParticipant::get_default_publisher_qos (p. 1163)
DDSPublisher::set_listener (p. 1368)

Examples:

HelloWorld_publisher.cxx.

6.184.2.27 virtual DDSPublisher* DDSDomainParticipant::create_publisher_with_profile (const char * *library_name*, const char * *profile_name*, DDSPublisherListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<eXtension>> (p. 199) Creates a new **DDSPublisher** (p. 1346) object using the **DDS_PublisherQos** (p. 851) associated with the input XML QoS profile.

Precondition:

The **DDS_PublisherQos** (p. 851) in the input profile must be consistent, or the operation will fail and no **DDSPublisher** (p. 1346) will be created.

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

listener *<<in>>* *(p. 200)*. Listener to be attached to the newly created **DDSPublisher** (p. 1346).

mask *<<in>>* *(p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

newly created publisher object or NULL on failure.

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_PublisherQos (p. 851) for rules on consistency among QoS
DDSDomainParticipant::create_publisher (p. 1169)
DDSDomainParticipant::get_default_publisher_qos (p. 1163)
DDSPublisher::set_listener (p. 1368)

6.184.2.28 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_publisher (DDSPublisher * p) [pure virtual]

Deletes an existing **DDSPublisher** (p. 1346).

Precondition:

The **DDSPublisher** (p. 1346) must not have any attached **DDSDataWriter** (p. 1113) objects. If there are existing **DDSDataWriter** (p. 1113) objects, it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).
DDSPublisher (p. 1346) must have been created by this **DDSDomainParticipant** (p. 1139), or else it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSPublisher** (p. 1346) will not be called after this method completes successfully.

Parameters:

p *<<in>>* *(p. 200)* **DDSPublisher** (p. 1346) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.184.2.29 virtual DDSSubscriber* DDSDomainParticipant::create_subscriber (const DDS_SubscriberQos & *qos*, DDSSubscriberListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

Creates a **DDSSubscriber** (p. 1390) with the desired QoS policies and attaches to it the specified **DDSSubscriberListener** (p. 1414).

Precondition:

The specified QoS policies must be consistent, or the operation will fail and no **DDSSubscriber** (p. 1390) will be created.

MT Safety:

UNSAFE. If **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) is used for `qos`, it is not safe to create the subscriber while another thread may be simultaneously calling **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167).

Parameters:

qos *<<in>> (p. 200)* QoS to be used for creating the new **DDSSubscriber** (p. 1390). The special value **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) can be used to indicate that the **DDSSubscriber** (p. 1390) should be created with the default **DDS_SubscriberQos** (p. 934) set in the **DDSDomainParticipant** (p. 1139).

listener *<<in>> (p. 200)*. Listener to be attached to the newly created **DDSSubscriber** (p. 1390).

mask *<<in>> (p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

newly created subscriber object or NULL on failure.

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_SubscriberQos (p. 934) for rules on consistency among QoS
DDS_SUBSCRIBER_QOS_DEFAULT (p. 39)
DDSDomainParticipant::create_subscriber_with_profile (p. 1173)
DDSDomainParticipant::get_default_subscriber_qos (p. 1166)
DDSSubscriber::set_listener (p. 1411)

Examples:

HelloWorld_subscriber.cxx.

6.184.2.30 virtual DDSSubscriber* DDSDomainParticipant::create_subscriber_with_profile (const char * *library_name*, const char * *profile_name*, DDSSubscriberListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<*eXtension*>> *(p. 199)* Creates a new **DDSSubscriber** (p. 1390) object using the **DDS_PublisherQos** (p. 851) associated with the input XML QoS profile.

Precondition:

The **DDS_SubscriberQos** (p. 934) in the input profile must be consistent, or the operation will fail and no **DDSSubscriber** (p. 1390) will be created.

Parameters:

library_name <<*in*>> *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name <<*in*>> *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

listener <<*in*>> *(p. 200)*. Listener to be attached to the newly created **DDSSubscriber** (p. 1390).

mask <<*in*>> *(p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

newly created subscriber object or NULL on failure.

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_SubscriberQos (p. 934) for rules on consistency among QoS
DDSDomainParticipant::create_subscriber (p. 1172)
DDSDomainParticipant::get_default_subscriber_qos (p. 1166)
DDSSubscriber::set_listener (p. 1411)

6.184.2.31 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_subscriber (DDSSubscriber * *s*) [pure virtual]

Deletes an existing **DDSSubscriber** (p. 1390).

Precondition:

The **DDSSubscriber** (p. 1390) must not have any attached **DDSDataReader** (p. 1087) objects. If there are existing **DDSDataReader** (p. 1087) objects, it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)
The **DDSSubscriber** (p. 1390) must have been created by this **DDSDomainParticipant** (p. 1139), or else it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

A Listener installed on the **DDSSubscriber** (p. 1390) will not be called after this method completes successfully.

Parameters:

s *<<in>> (p. 200)* **DDSSubscriber** (p. 1390) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.184.2.32 virtual DDS_ReturnCode_t DDSDomainParticipant::get_publishers (DDSPublisherSeq & *publishers*) [pure virtual]

<<eXtension>> (p. 199) Allows the application to access all the publishers the participant has.

If the sequence doesn't own its buffer, and its maximum is less than the total number of publishers, it will be filled up to its maximum, and fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

Safe.

Parameters:

publishers *<<inout>> (p. 200)* a PublisherSeq object where the set or list of publishers will be returned

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

6.184.2.33 virtual DDS_ReturnCode_t DDSDomainParticipant::get_subscribers (DDSSubscriberSeq & *subscribers*) [pure virtual]

<<***eXtension***>> *(*p. *199)* Allows the application to access all the subscribers the participant has.

If the sequence doesn't own its buffer, and its maximum is less than the total number of subscribers, it will be filled up to its maximum, and fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

Safe.

Parameters:

subscribers <<***inout***>> *(*p. *200)* a SubscriberSeq object where the set or list of subscribers will be returned

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315)

6.184.2.34 virtual DDSTopic* DDSDomainParticipant::create_topic (const char * *topic_name*, const char * *type_name*, const DDS_TopicQos & *qos*, DDSTopicListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

Creates a **DDSTopic** (p. 1419) with the desired QoS policies and attaches to it the specified **DDSTopicListener** (p. 1430).

Precondition:

The application is not allowed to create two **DDSTopic** (p. 1419) objects with the same `topic_name` attached to the same **DDSDomainParticipant** (p. 1139). If the application attempts this, this method will fail and return a NULL topic.
The specified QoS policies must be consistent, or the operation will fail and no **DDSTopic** (p. 1419) will be created.
Prior to creating a **DDSTopic** (p. 1419), the type must have been registered with RTI Connext. This is done using the **FooTypeSupport::register_type** (p. 1510) operation on a derived class of the **DDSTypeSupport** (p. 1432) interface.

MT Safety:

UNSAFE. It is not safe to create a topic while another thread is trying to lookup that topic description with **DDSDomainParticipant::lookup_topicdescription** (p. 1183).

MT Safety:

UNSAFE. If **DDS_TOPIC_QOS_DEFAULT** (p. 38) is used for `qos`, it is not safe to create the topic while another thread may be simultaneously calling **DDSDomainParticipant::set_default_topic_qos** (p. 1162).

Parameters:

topic_name *<<in>> (p. 200)* Name for the new topic, must not exceed 255 characters. Cannot be NULL.

type_name *<<in>> (p. 200)* The type to which the new **DDSTopic** (p. 1419) will be bound. Cannot be NULL.

qos *<<in>> (p. 200)* QoS to be used for creating the new **DDSTopic** (p. 1419). The special value **DDS_TOPIC_QOS_DEFAULT** (p. 38) can be used to indicate that the **DDSTopic** (p. 1419) should be created with the default **DDS_TopicQos** (p. 965) set in the **DDSDomainParticipant** (p. 1139).

listener *<<in>> (p. 200)*. Listener to be attached to the newly created **DDSTopic** (p. 1419).

mask *<<in>> (p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

newly created topic, or NULL on failure

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_TopicQos (p. 965) for rules on consistency among QoS
DDS_TOPIC_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_topic_with_profile (p. 1177)
DDSDomainParticipant::get_default_topic_qos (p. 1161)
DDSTopic::set_listener (p. 1423)

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

6.184.2.35 virtual DDSTopic* DDSDomainParticipant::create_topic_with_profile (const char * *topic_name*, const char * *type_name*, const char * *library_name*, const char * *profile_name*, DDSTopicListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<*eXtension*>> *(p. 199)* Creates a new **DDSTopic** (p. 1419) object using the **DDS_PublisherQos** (p. 851) associated with the input XML QoS profile.

Precondition:

The application is not allowed to create two **DDSTopicDescription** (p. 1427) objects with the same `topic_name` attached to the same **DDSDomainParticipant** (p. 1139). If the application attempts this, this method will fail and return a NULL topic.
The **DDS_TopicQos** (p. 965) in the input profile must be consistent, or the operation will fail and no **DDSTopic** (p. 1419) will be created.
Prior to creating a **DDSTopic** (p. 1419), the type must have been registered with RTI Connext. This is done using the **FooTypeSupport::register_type** (p. 1510) operation on a derived class of the **DDSTypeSupport** (p. 1432) interface.

MT Safety:

UNSAFE. It is not safe to create a topic while another thread is trying to lookup that topic description with **DDSDomainParticipant::lookup_topicdescription** (p. 1183).

Parameters:

topic_name <<*in*>> *(p. 200)* Name for the new topic, must not exceed 255 characters. Cannot be NULL.

type_name <<*in*>> *(p. 200)* The type to which the new **DDSTopic** (p. 1419) will be bound. Cannot be NULL.

library_name <<*in*>> *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name <<*in*>> *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

listener <<*in*>> *(p. 200)*. Listener to be attached to the newly created **DDSTopic** (p. 1419).

mask <<*in*>> *(p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

newly created topic, or NULL on failure

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_TopicQos (p. 965) for rules on consistency among QoS
DDSDomainParticipant::create_topic (p. 1175)
DDSDomainParticipant::get_default_topic_qos (p. 1161)
DDSTopic::set_listener (p. 1423)

6.184.2.36 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_topic (DDSTopic * *topic*) [pure virtual]

Deletes a **DDSTopic** (p. 1419).

Precondition:

If the **DDSTopic** (p. 1419) does not belong to the application's **DDSDomainParticipant** (p. 1139), this operation fails with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).
Make sure no objects are using the topic. More specifically, there must be no existing **DDSDataReader** (p. 1087), **DDSDataWriter** (p. 1113), **DDSContentFilteredTopic** (p. 1081), or **DDSMultiTopic** (p. 1322) objects belonging to the same **DDSDomainParticipant** (p. 1139) that are using the **DDSTopic** (p. 1419). If delete_topic is called on a **DDSTopic** (p. 1419) with any of these existing objects attached to it, it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSTopic** (p. 1419) will not be called after this method completes successfully.

Parameters:

topic *<<in>>* *(p. 200)* **DDSTopic** (p. 1419) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

6.184.2.37 virtual DDSContentFilteredTopic* DDSDomainParticipant::create_contentfilteredtopic (const char * *name*, DDSTopic * *related_topic*, const char * *filter_expression*, const DDS_StringSeq & *expression_parameters*) [pure virtual]

Creates a **DDSContentFilteredTopic** (p. 1081), that can be used to do content-based subscriptions.

The **DDSContentFilteredTopic** (p. 1081) only relates to samples published under that **DDSTopic** (p. 1419), filtered according to their content. The filtering is done by means of evaluating a logical expression that involves the values of some of the data-fields in the sample. The logical expression derived from the `filter_expression` and `expression_parameters` arguments.

Queries and Filters Syntax (p. 208) describes the syntax of `filter_expression` and `expression_parameters`.

Precondition:

The application is not allowed to create two **DDSContentFilteredTopic** (p. 1081) objects with the same `topic_name` attached to the same **DDSDomainParticipant** (p. 1139). If the application attempts this, this method will fail and returns NULL.
If `related_topic` does not belong to this **DDSDomainParticipant** (p. 1139), this operation returns NULL.
This function will create a content filter using the builtin SQL filter which implements a superset of the DDS specification. This filter **requires** that all IDL types have been compiled with typecodes. If this precondition is not met, this operation returns NULL. Do not use rtiddsgen's -notypecode option if you want to use the builtin SQL filter.

Parameters:

name *<<in>>* *(p. 200)* Name for the new content filtered topic, must not exceed 255 characters. Cannot be NULL.

related_topic *<<in>>* *(p. 200)* **DDSTopic** (p. 1419) to be filtered. Cannot be NULL.

filter_expression *<<in>>* *(p. 200)* Cannot be NULL

expression_parameters *<<in>>* *(p. 200)* An empty sequence **must** be used if the filter expression does not contain any parameters. Length of sequence cannot be greater than 100.

Returns:

newly created **DDSContentFilteredTopic** (p. 1081), or NULL on failure

6.184.2.38 virtual DDSContentFilteredTopic* DDSDomainParticipant::create_contentfilteredtopic_with_filter (const char * *name*, DDSTopic * *related_topic*, const char * *filter_expression*, const DDS_StringSeq & *expression_parameters*, const char * *filter_name* = DDS_SQLFILTER_NAME) [pure virtual]

<<eXtension>> *(p. 199)* Creates a **DDSContentFilteredTopic** (p. 1081) using the specified filter to do content-based subscriptions.

Parameters:

name *<<in>>* *(p. 200)* Name for the new content filtered topic. Cannot exceed 255 characters. Cannot be NULL.

related_topic *<<in>>* *(p. 200)* **DDSTopic** (p. 1419) to be filtered. Cannot be NULL.

filter_expression *<<in>>* *(p. 200)* Cannot be NULL.

expression_parameters *<<in>>* *(p. 200)* . An empty sequence **must** be used if the filter expression does not contain any parameters. Length of the sequence cannot be greater than 100.

filter_name *<<in>>* *(p. 200)* Name of content filter to use. Must previously have been registered with **DDSDomainParticipant::register_contentfilter** (p. 1156) on the same **DDSDomainParticipant** (p. 1139). Cannot be NULL.

Builtin filter names are **DDS_SQLFILTER_NAME** (p. 41) and **DDS_STRINGMATCHFILTER_NAME** (p. 41)

Returns:

newly created **DDSContentFilteredTopic** (p. 1081), or NULL on failure

6.184.2.39 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_contentfilteredtopic (DDSContentFilteredTopic * *a_contentfilteredtopic*) [pure virtual]

Deletes a **DDSContentFilteredTopic** (p. 1081).

Precondition:

The deletion of a **DDSContentFilteredTopic** (p. 1081) is not allowed if there are any existing **DDSDataReader** (p. 1087) objects that are using

the **DDSContentFilteredTopic** (p. 1081). If the operation is called on a **DDSContentFilteredTopic** (p. 1081) with existing **DDSDataReader** (p. 1087) objects attached to it, it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).
The **DDSContentFilteredTopic** (p. 1081) must be created by this **DDSDomainParticipant** (p. 1139), or else this operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Parameters:

a_contentfilteredtopic *<<in>> (p. 200)*

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

6.184.2.40 virtual DDSMultiTopic* DDSDomainParticipant::create_multitopic (const char * *name*, const char * *type_name*, const char * *subscription_expression*, const DDS_StringSeq & *expression_parameters*) [pure virtual]

[**Not supported (optional)**] Creates a MultiTopic that can be used to subscribe to multiple topics and combine/filter the received data into a resulting type.

The resulting type is specified by the `type_name` argument. The list of topics and the logic used to combine, filter, and rearrange the information from each **DDSTopic** (p. 1419) are specified using the `subscription_expression` and `expression_parameters` arguments.

Queries and Filters Syntax (p. 208) describes the syntax of `subscription_expression` and `expression_parameters`.

Precondition:

The application is not allowed to create two **DDSTopicDescription** (p. 1427) objects with the same `name` attached to the same **DDSDomainParticipant** (p. 1139). If the application attempts this, this method will fail and return NULL.
Prior to creating a **DDSMultiTopic** (p. 1322), the type must have been registered with RTI Connext. This is done using the **FooTypeSupport::register_type** (p. 1510) operation on a derived class of the **DDSTypeSupport** (p. 1432) interface. Otherwise, this method will return NULL.

Parameters:

name *<<in>>* *(p. 200)* Name of the newly create **DDSMultiTopic** (p. 1322). Cannot be NULL.

type_name *<<in>>* *(p. 200)* Cannot be NULL.

subscription_expression *<<in>>* *(p. 200)* Cannot be NULL.

expression_parameters *<<in>>* *(p. 200)*

Returns:

NULL

6.184.2.41 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_multitopic (DDSMultiTopic * *a_multitopic*) [pure virtual]

[**Not supported (optional)**] Deletes a **DDSMultiTopic** (p. 1322).

Precondition:

The deletion of a **DDSMultiTopic** (p. 1322) is not allowed if there are any existing **DDSDataReader** (p. 1087) objects that are using the **DDSMultiTopic** (p. 1322). If the delete_multitopic operation is called on a **DDSMultiTopic** (p. 1322) with existing **DDSDataReader** (p. 1087) objects attached to it, it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).
The **DDSMultiTopic** (p. 1322) must be created by this **DDSDomainParticipant** (p. 1139), or else this operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Parameters:

a_multitopic *<<in>>* *(p. 200)*

Returns:

DDS_RETCODE_UNSUPPORTED (p. 315)

6.184.2.42 virtual DDSTopic* DDSDomainParticipant::find_topic (const char * *topic_name*, const DDS_Duration_t & *timeout*) [pure virtual]

Finds an existing (or ready to exist) **DDSTopic** (p. 1419), based on its name.

This call can be used to block for a specified duration to wait for the **DDSTopic** (p. 1419) to be created.

If the requested **DDSTopic** (p. 1419) already exists, it is returned. Otherwise, **find_topic()** (p. 1182) waits until another thread creates it or else returns when the specified timeout occurs.

find_topic() (p. 1182) is useful when multiple threads are concurrently creating and looking up topics. In that case, one thread can call **find_topic()** (p. 1182) and, if another thread has not yet created the topic being looked up, it can wait for some period of time for it to do so. In almost all other cases, it is more straightforward to call **DDSDomainParticipant::lookup_topicdescription** (p. 1183).

The **DDSDomainParticipant** (p. 1139) must already be enabled.

Note: *Each* **DDSTopic** (p. 1419) obtained by **DDSDomainParticipant::find_topic** (p. 1182) must also be deleted by means of **DDSDomainParticipant::delete_topic** (p. 1178). If **DDSTopic** (p. 1419) is obtained multiple times by means of **DDSDomainParticipant::find_topic** (p. 1182) or **DDSDomainParticipant::create_topic** (p. 1175), it must also be deleted that same number of times using **DDSDomainParticipant::delete_topic** (p. 1178).

Parameters:

topic_name *<<in>> (p. 200)* Name of the **DDSTopic** (p. 1419) to search for. Cannot be NULL.

timeout *<<in>> (p. 200)* The time to wait if the **DDSTopic** (p. 1419) does not exist already.

Returns:

the topic, if it exists, or NULL

6.184.2.43 virtual DDSTopicDescription* DDSDomainParticipant::lookup_topicdescription (const char * *topic_name*) [pure virtual]

Looks up an existing, locally created **DDSTopicDescription** (p. 1427), based on its name.

DDSTopicDescription (p. 1427) is the base class for **DDSTopic** (p. 1419), **DDSMultiTopic** (p. 1322) and **DDSContentFilteredTopic** (p. 1081). So you can narrow the **DDSTopicDescription** (p. 1427) returned from this operation to a **DDSTopic** (p. 1419) or **DDSContentFilteredTopic** (p. 1081) as appropriate.

Unlike **DDSDomainParticipant::find_topic** (p. 1182), which logically returns a new **DDSTopic** (p. 1419) object that must be independently deleted, *this* operation returns a reference to the original local object.

The **DDSDomainParticipant** (p. 1139) does not have to be enabled when you call **lookup_topicdescription()** (p. 1183).

The returned topic may be either enabled or disabled.

MT Safety:

UNSAFE. It is not safe to lookup a topic description while another thread is creating that topic.

Parameters:

topic_name *<<in>>* *(p. 200)* Name of **DDSTopicDescription** (p. 1427) to search for. This string must be no more than 255 characters; it cannot be NULL.

Returns:

The topic description, if it has already been created locally, otherwise it returns NULL.

6.184.2.44 virtual DDSFlowController* DDSDomainParticipant::create_flowcontroller (const char * *name*, const DDS_FlowControllerProperty_t & *prop*) [pure virtual]

<<eXtension>> *(p. 199)* Creates a **DDSFlowController** (p. 1259) with the desired property.

The created **DDSFlowController** (p. 1259) is associated with a **DDSDataWriter** (p. 1113) via **DDS_PublishModeQosPolicy::flow_controller_name** (p. 855). A single **DDSFlowController** (p. 1259) may service multiple **DDSDataWriter** (p. 1113) instances, even if they belong to a different **DDSPublisher** (p. 1346). The `property` determines how the **DDSFlowController** (p. 1259) shapes the network traffic.

Precondition:

The specified `property` must be consistent, or the operation will fail and no **DDSFlowController** (p. 1259) will be created.

MT Safety:

UNSAFE. If **DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT** (p. 40) is used for `property`, it is not safe to create the

flow controller while another thread may be simultaneously calling **DDSDomainParticipant::set_default_flowcontroller_property** (p. 1155) or trying to lookup that flow controller with **DDSDomainParticipant::lookup_flowcontroller** (p. 1186).

Parameters:

name *<<in>>* *(p. 200)* name of the **DDSFlowController** (p. 1259) to create. A **DDSDataWriter** (p. 1113) is associated with a **DDSFlowController** (p. 1259) by name. Limited to 255 characters.

prop *<<in>>* *(p. 200)* property to be used for creating the new **DDSFlowController** (p. 1259). The special value **DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT** (p. 40) can be used to indicate that the **DDSFlowController** (p. 1259) should be created with the default **DDS_FlowControllerProperty_t** (p. 749) set in the **DDSDomainParticipant** (p. 1139).

Returns:

Newly created flow controller object or NULL on failure.

See also:

DDS_FlowControllerProperty_t (p. 749) for rules on consistency among property
DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT (p. 40)
DDSDomainParticipant::get_default_flowcontroller_property (p. 1154)

6.184.2.45 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_flowcontroller (DDSFlowController * *fc*) [pure virtual]

<<eXtension>> *(p. 199)* Deletes an existing **DDSFlowController** (p. 1259).

Precondition:

The **DDSFlowController** (p. 1259) must not have any attached **DDSDataWriter** (p. 1113) objects. If there are any attached **DDSDataWriter** (p. 1113) objects, it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).
The **DDSFlowController** (p. 1259) must have been created by this **DDSDomainParticipant** (p. 1139), or else it will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

The **DDSFlowController** (p. 1259) is deleted if this method completes successfully.

Parameters:

fc *<<**in**>>* *(p. 200)* The **DDSFlowController** (p. 1259) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.184.2.46 virtual DDSFlowController* DDSDomainParticipant::lookup_flowcontroller (const char * *name*) [pure virtual]

*<<**eXtension**>>* *(p. 199)* Looks up an existing locally-created **DDSFlowController** (p. 1259), based on its name.

Looks up a previously created **DDSFlowController** (p. 1259), including the built-in ones. Once a **DDSFlowController** (p. 1259) has been deleted, subsequent lookups will fail.

MT Safety:

UNSAFE. It is not safe to lookup a flow controller description while another thread is creating that flow controller.

Parameters:

name *<<**in**>>* *(p. 200)* Name of **DDSFlowController** (p. 1259) to search for. Limited to 255 characters. Cannot be NULL.

Returns:

The flow controller if it has already been created locally, or NULL otherwise.

6.184.2.47 virtual DDSSubscriber* DDSDomainParticipant::get_builtin_subscriber () [pure virtual]

Accesses the **built-in DDSSubscriber** (p. 1390).

Each **DDSDomainParticipant** (p. 1139) contains several built-in **DDSTopic** (p. 1419) objects as well as corresponding **DDSDataReader** (p. 1087) objects

to access them. All of these **DDSDataReader** (p. 1087) objects belong to a single built-in **DDSSubscriber** (p. 1390).

The built-in Topics are used to communicate information about other **DDSDomainParticipant** (p. 1139), **DDSTopic** (p. 1419), **DDSDataReader** (p. 1087), and **DDSDataWriter** (p. 1113) objects.

The built-in subscriber is created when this operation is called for the first time. The built-in subscriber is deleted automatically when the **DDSDomainParticipant** (p. 1139) is deleted.

Returns:

The built-in **DDSSubscriber** (p. 1390) singleton.

See also:

DDS_SubscriptionBuiltinTopicData (p. 936)
DDS_PublicationBuiltinTopicData (p. 839)
DDS_ParticipantBuiltinTopicData (p. 816)
DDS_TopicBuiltinTopicData (p. 958)

6.184.2.48 virtual DDS_ReturnCode_t DDSDomainParticipant::ignore_participant (const DDS_InstanceHandle_t & *handle*) [pure virtual]

Instructs RTI Connext to locally ignore a remote **DDSDomainParticipant** (p. 1139).

From the time of this call onwards, RTI Connext will locally behave as if the remote participant did not exist. This means it will ignore any topic, publication, or subscription that originates on that **DDSDomainParticipant** (p. 1139).

There is no way to reverse this operation.

This operation can be used in conjunction with the discovery of remote participants offered by means of the **DDS_ParticipantBuiltinTopicData** (p. 816) to provide access control.

Application data can be associated with a **DDSDomainParticipant** (p. 1139) by means of the **USER_DATA** (p. 345) policy. This application data is propagated as a field in the built-in topic and can be used by an application to implement its own access control policy.

The **DDSDomainParticipant** (p. 1139) to ignore is identified by the `handle` argument. This `handle` is the one that appears in the **DDS_SampleInfo** (p. 912) retrieved when reading the data-samples available for the built-in **DDSDataReader** (p. 1087) to the `DDSDomainParticipant` (p. 1139) topic.

The built-in **DDSDataReader** (p. 1087) is read with the same **FooDataReader::read** (p. 1447) and **FooDataReader::take** (p. 1448) operations used for any **DDSDataReader** (p. 1087).

Parameters:

handle *<<in>> (p. 200)* **DDS_InstanceHandle_t** (p. 53) of the **DDSDomainParticipant** (p. 1139) to be ignored.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315), **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_ParticipantBuiltinTopicData (p. 816)
DDS_PARTICIPANT_TOPIC_NAME (p. 285)
DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.184.2.49 virtual DDS_ReturnCode_t DDSDomainParticipant::ignore_topic (const DDS_InstanceHandle_t & *handle*) [pure virtual]

Instructs RTI Connext to locally ignore a **DDSTopic** (p. 1419).

This means it will locally ignore any publication, or subscription to the **DDSTopic** (p. 1419).

There is no way to reverse this operation.

This operation can be used to save local resources when the application knows that it will never publish or subscribe to data under certain topics.

The **DDSTopic** (p. 1419) to ignore is identified by the `handle` argument. This is the handle of a **DDSTopic** (p. 1419) that appears in the **DDS_SampleInfo** (p. 912) retrieved when reading data samples from the built-in **DDSDataReader** (p. 1087) for the **DDSTopic** (p. 1419).

Parameters:

handle *<<in>> (p. 200)* Handle of the **DDSTopic** (p. 1419) to be ignored.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_TopicBuiltinTopicData (p. 958)
DDS_TOPIC_TOPIC_NAME (p. 287)
DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.184.2.50 virtual DDS_ReturnCode_t DDSDomainParticipant::ignore_publication (const DDS_InstanceHandle_t & *handle*) [pure virtual]

Instructs RTI Connext to locally ignore a publication.

A publication is defined by the association of a topic name, user data, and partition set on the **DDSPublisher** (p. 1346) (see **DDS_PublicationBuiltinTopicData** (p. 839)). After this call, any data written by that publication's **DDSDataWriter** (p. 1113) will be ignored.

This operation can be used to ignore local *and* remote DataWriters.

The publication (DataWriter) to ignore is identified by the `handle` argument.

- To ignore a *remote* DataWriter, the `handle` can be obtained from the **DDS_SampleInfo** (p. 912) retrieved when reading data samples from the built-in **DDSDataReader** (p. 1087) for the publication topic.

- To ignore a *local* DataWriter, the `handle` can be obtained by calling **DDSEntity::get_instance_handle** (p. 1258) for the local DataWriter.

There is no way to reverse this operation.

Parameters:

handle *<<in>>* *(p. 200)* Handle of the **DDSDataWriter** (p. 1113) to be ignored.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_PublicationBuiltinTopicData (p. 839)
DDS_PUBLICATION_TOPIC_NAME (p. 289)
DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.184.2.51 virtual DDS_ReturnCode_t DDSDomainParticipant::ignore_subscription (const DDS_InstanceHandle_t & *handle*) [pure virtual]

Instructs RTI Connext to locally ignore a subscription.

A subscription is defined by the association of a topic name, user data, and partition set on the **DDSSubscriber** (p. 1390) (see **DDS_SubscriptionBuiltinTopicData** (p. 936)). After this call, any data received related to that subscription's **DDSDataReader** (p. 1087) will be ignored.

This operation can be used to ignore local *and* remote DataReaders.

The subscription to ignore is identified by the `handle` argument.

- To ignore a *remote* DataReader, the `handle` can be obtained from the **DDS_SampleInfo** (p. 912) retrieved when reading data samples from the built-in **DDSDataReader** (p. 1087) for the subscription topic.

- To ignore a *local* DataReader, the `handle` can be obtained by calling **DDSEntity::get_instance_handle** (p. 1258) for the local DataReader.

There is no way to reverse this operation.

Parameters:

handle *<<in>> (p. 200)* Handle of the **DDSDataReader** (p. 1087) to be ignored.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_SubscriptionBuiltinTopicData (p. 936)
DDS_SUBSCRIPTION_TOPIC_NAME (p. 291)
DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.184.2.52 virtual DDS_DomainId_t DDSDomainParticipant::get_domain_id () [pure virtual]

Get the unique domain identifier.

This operation retrieves the `domain_id` used to create the **DDSDomainParticipant** (p. 1139). The `domain_id` identifies the DDS domain to which the

DDSDomainParticipant (p. 1139) belongs. Each DDS domain represents a separate data 'communication plane' isolated from other domains.

Returns:

the unique `domainId` that was used to create the domain

See also:

DDSDomainParticipantFactory::create_participant (p. 1233)
DDSDomainParticipantFactory::create_participant_with_profile (p. 1235)

6.184.2.53 virtual DDS_ReturnCode_t DDSDomainParticipant::get_current_time (DDS_Time_t & *current_time*) [pure virtual]

Returns the current value of the time.

The current value of the time that RTI Connext uses to time-stamp **DDSDataWriter** (p. 1113) and to set the reception-timestamp for the data updates that it receives.

Parameters:

current_time <<***inout***>> *(p. 200)* Current time to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

6.184.2.54 virtual DDS_ReturnCode_t DDSDomainParticipant::register_durable_subscription (DDS_EndpointGroup_t & *group*, const char * *topic_name*) [pure virtual]

Registers a Durable Subscription on the specified **DDSTopic** (p. 1419) on all Persistence Services.

If you need to receive all samples published on a **DDSTopic** (p. 1419), including the ones published while a **DDSDataReader** (p. 1087) is inactive or before it may be created, create a Durable Subscription using this method.

In this way, the Persistence Service will ensure that all the samples on that **DDSTopic** (p. 1419) are retained until they are acknowledged by at least N DataReaders belonging to the Durable Subscription where N is the quorum count.

If the same Durable Subscription is created on a different **DDSTopic** (p. 1419), the Persistence Service will implicitly delete the previous Durable Subscription and create a new one on the new **DDSTopic** (p. 1419).

Parameters:

group *<<in>>* *(p. 200)* **DDS_EndpointGroup_t** (p. 731) The Durable Subscription name and quorum.

topic_name *<<in>>* *(p. 200)* The topic name for which the Durable Subscription is created.

Returns:

One of the **Standard Return Codes** (p. 314)

6.184.2.55 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_durable_subscription (DDS_EndpointGroup_t & *group*) [pure virtual]

Deletes an existing Durable Subscription on all Persistence Services.

The Persistence Service will delete the Durable Subscription and the quorum of the existing samples will be considered satisfied.

Parameters:

group *<<in>>* *(p. 200)* **DDS_EndpointGroup_t** (p. 731) specifying the Durable Subscription name. Quorum is not required for this operation.

Returns:

One of the **Standard Return Codes** (p. 314)

6.184.2.56 virtual DDS_ReturnCode_t DDSDomainParticipant::assert_liveliness () [pure virtual]

Manually asserts the liveliness of this **DDSDomainParticipant** (p. 1139).

This is used in combination with the **DDS_LivelinessQosPolicy** (p. 779) to indicate to RTI Connext that the entity remains active.

You need to use this operation if the **DDSDomainParticipant** (p. 1139) conatins **DDSDataWriter** (p. 1113) entities with the **DDS_LivelinessQosPolicy::kind** (p. 782) set to **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** (p. 359) and it only affects the

liveliness of those **DDSDataWriter** (p. 1113) entities. Otherwise, it has no effect.

Note: writing data via the **FooDataWriter::write** (p. 1484) or **FooDataWriter::write_w_timestamp** (p. 1486) operation asserts liveliness on the **DDSDataWriter** (p. 1113) itself and its **DDSDomainParticipant** (p. 1139). Consequently the use of **assert_liveliness()** (p. 1192) is only needed if the application is not writing data regularly.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_LivelinessQosPolicy (p. 779)

6.184.2.57 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_contained_entities () [pure virtual]

Delete all the entities that were created by means of the "create" operations on the **DDSDomainParticipant** (p. 1139).

This operation deletes all contained **DDSPublisher** (p. 1346) (including an implicit Publisher, if one exists), **DDSSubscriber** (p. 1390) (including implicit subscriber), **DDSTopic** (p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322) objects.

Prior to deleting each contained entity, this operation will recursively call the corresponding delete_contained_entities operation on each contained entity (if applicable). This pattern is applied recursively. In this manner the operation **delete_contained_entities()** (p. 1193) on the **DDSDomainParticipant** (p. 1139) will end up deleting all the entities recursively contained in the **DDSDomainParticipant** (p. 1139), that is also the **DDSDataWriter** (p. 1113), **DDSDataReader** (p. 1087), as well as the **DDSQueryCondition** (p. 1372) and **DDSReadCondition** (p. 1374) objects belonging to the contained **DDSDataReader** (p. 1087).

The operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) if any of the contained entities is in a state where it cannot be deleted.

If **delete_contained_entities()** (p. 1193) completes successfully, the application may delete the **DDSDomainParticipant** (p. 1139) knowing that it has no contained entities.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_-PRECONDITION_NOT_MET** (p. 315).

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

6.184.2.58 virtual DDS_ReturnCode_t DDSDomainParticipant::get_discovered_participants (DDS_InstanceHandleSeq & *participant_handles*) [pure virtual]

Returns list of discovered **DDSDomainParticipant** (p. 1139) s.

This operation retrieves the list of **DDSDomainParticipant** (p. 1139) s that have been discovered in the domain and that the application has not indicated should be "ignored" by means of the **DDSDomainParticipant::ignore_-participant** (p. 1187) operation.

Parameters:

participant_handles *<<inout>>* *(p. 200)* DDSInstanceHandleSeq to be filled with handles of the discovered **DDSDomainParticipant** (p. 1139) s

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_-NOT_ENABLED** (p. 315)

6.184.2.59 virtual DDS_ReturnCode_t DDSDomainParticipant::get_discovered_participant_data (struct DDS_ParticipantBuiltinTopicData & *participant_data*, const DDS_InstanceHandle_t & *participant_handle*) [pure virtual]

Returns **DDS_ParticipantBuiltinTopicData** (p. 816) for the specified **DDS-DomainParticipant** (p. 1139) .

This operation retrieves information on a **DDSDomainParticipant** (p. 1139) that has been discovered on the network. The participant must be in the same domain as the participant on which this operation is invoked and must not have been "ignored" by means of the **DDSDomainParticipant::ignore_-participant** (p. 1187) operation.

The `participant_handle` must correspond to such a DomainParticipant. Otherwise, the operation will fail with PRECONDITION_NOT_MET.

Use the operation **DDSDomainParticipant::get_discovered_participants** (p. 1194) to find the **DDSDomainParticipant** (p. 1139) s that are currently discovered.

Note: This operation does not retrieve the **DDS_ParticipantBuiltinTopicData::property** (p. 817). This information is available through **DDSDataReaderListener::on_data_available()** (p. 1110) (if a reader listener is installed on the **DDSParticipantBuiltinTopicDataDataReader** (p. 1341)).

Parameters:

participant_data *<<inout>>* (p. *200*) **DDS_ParticipantBuiltinTopicData** (p. 816) to be filled with the specified **DDSDomainParticipant** (p. 1139) 's data.

participant_handle *<<in>>* (p. *200*) **DDS_InstanceHandle_t** (p. 53) of **DDSDomainParticipant** (p. 1139).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_ParticipantBuiltinTopicData (p. 816)
DDSDomainParticipant::get_discovered_participants (p. 1194)

6.184.2.60 virtual DDS_ReturnCode_t DDSDomainParticipant::get_discovered_topics (DDS_InstanceHandleSeq & *topic_handles*) [pure virtual]

Returns list of discovered **DDSTopic** (p. 1419) objects.

This operation retrieves the list of **DDSTopic** (p. 1419) s that have been discovered in the domain and that the application has not indicated should be "ignored" by means of the **DDSDomainParticipant::ignore_topic** (p. 1188) operation.

Parameters:

topic_handles *<<inout>>* (p. *200*) DDSInstanceHandleSeq to be filled with handles of the discovered **DDSTopic** (p. 1419) objects

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

6.184.2.61 virtual DDS_ReturnCode_t DDSDomainParticipant::get_discovered_topic_data (struct DDS_TopicBuiltinTopicData & *topic_data*, const DDS_InstanceHandle_t & *topic_handle*) [pure virtual]

Returns **DDS_TopicBuiltinTopicData** (p. 958) for the specified **DDSTopic** (p. 1419).

This operation retrieves information on a **DDSTopic** (p. 1419) that has been discovered by the local Participant and must not have been "ignored" by means of the **DDSDomainParticipant::ignore_topic** (p. 1188) operation.

The `topic_handle` must correspond to such a topic. Otherwise, the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

This call is not supported for remote topics. If a remote `topic_handle` is used, the operation will fail with **DDS_RETCODE_UNSUPPORTED** (p. 315).

Use the operation **DDSDomainParticipant::get_discovered_topics** (p. 1195) to find the topics that are currently discovered.

Parameters:

topic_data *<<inout>>* *(p. 200)* **DDS_TopicBuiltinTopicData** (p. 958) to be filled with the specified **DDSTopic** (p. 1419)'s data.

topic_handle *<<in>>* *(p. 200)* **DDS_InstanceHandle_t** (p. 53) of **DDSTopic** (p. 1419).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

DDS_TopicBuiltinTopicData (p. 958)
DDSDomainParticipant::get_discovered_topics (p. 1195)

6.184.2.62 virtual DDS_Boolean DDSDomainParticipant::contains_entity (const DDS_InstanceHandle_t & *a_handle*) [pure virtual]

Completes successfully with **DDS_BOOLEAN_TRUE** (p. 298) if the referenced **DDSEntity** (p. 1253) is contained by the **DDSDomainParticipant** (p. 1139).

This operation checks whether or not the given `a_handle` represents an **DDSEntity** (p. 1253) that was created from the **DDSDomainParticipant** (p. 1139). The containment applies recursively. That is, it applies both to entities (**DDSTopicDescription** (p. 1427), **DDSPublisher** (p. 1346), or **DDSSubscriber** (p. 1390)) created directly using the **DDSDomainParticipant** (p. 1139) as well as entities created using a contained **DDSPublisher** (p. 1346), or **DDSSubscriber** (p. 1390) as the factory, and so forth.

The `instance` handle for an **DDSEntity** (p. 1253) may be obtained from built-in topic data, from various statuses, or from the operation **DDSEntity::get_instance_handle** (p. 1258).

Parameters:

a_handle *<<in>> (p. 200)* **DDS_InstanceHandle_t** (p. 53) of the **DDSEntity** (p. 1253) to be checked.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if **DDSEntity** (p. 1253) is contained by the **DDSDomainParticipant** (p. 1139), or **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

6.184.2.63 virtual DDS_ReturnCode_t DDSDomainParticipant::set_qos (const DDS_DomainParticipantQos & *qos*) [pure virtual]

Change the QoS of this DomainParticipant.

The **DDS_DomainParticipantQos::user_data** (p. 590) and **DDS_DomainParticipantQos::entity_factory** (p. 590) can be changed. The other policies are immutable.

Parameters:

qos *<<in>> (p. 200)* Set of policies to be applied to **DDSDomainParticipant** (p. 1139). Policies must be consistent. Immutable policies cannot be changed after **DDSDomainParticipant** (p. 1139) is enabled. The special value **DDS_PARTICIPANT_QOS_DEFAULT**

(p. 35) can be used to indicate that the QoS of the **DDSDomainParticipant** (p. 1139) should be changed to match the current default **DDS_DomainParticipantQos** (p. 588) set in the **DDSDomainParticipantFactory** (p. 1216).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) if immutable policy is changed, or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) if policies are inconsistent

See also:

DDS_DomainParticipantQos (p. 588) for rules on consistency among QoS
set_qos (abstract) (p. 1254)

6.184.2.64 virtual DDS_ReturnCode_t DDSDomainParticipant::set_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<***eXtension***>> *(p. 199)* Change the QoS of this domain participant using the input XML QoS profile.

The **DDS_DomainParticipantQos::user_data** (p. 590) and **DDS_DomainParticipantQos::entity_factory** (p. 590) can be changed. The other policies are immutable.

Parameters:

library_name <<***in***>> *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name <<***in***>> *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) if immutable policy is changed, or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) if policies are inconsistent

See also:

DDS_DomainParticipantQos (p. 588) for rules on consistency among QoS

6.184.2.65 virtual DDS_ReturnCode_t DDSDomainParticipant::get_qos (DDS_DomainParticipantQos & *qos*) [pure virtual]

Get the participant QoS.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos <<***inout***>> *(p. 200)* QoS to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

get_qos (abstract) (p. 1255)

6.184.2.66 virtual DDS_ReturnCode_t DDSDomainParticipant::add_peer (const char * *peer_desc_string*) [pure virtual]

<<***eXtension***>> *(p. 199)* Attempt to contact one or more additional peer participants.

Add the given peer description to the list of peers with which this **DDSDomainParticipant** (p. 1139) will try to communicate.

This method may be called at any time after this **DDSDomainParticipant** (p. 1139) has been created (before or after it has been enabled).

If this method is called after **DDSEntity::enable** (p. 1256), an attempt will be made to contact the new peer(s) immediately.

If this method is called *before* the DomainParticipant is enabled, the peer description will simply be added to the list that was populated by **DDS_DiscoveryQosPolicy::initial_peers** (p. 583); the first attempted contact will take place after this **DDSDomainParticipant** (p. 1139) is enabled.

Adding a peer description with this method does not guarantee that any peer(s) discovered as a result will exactly correspond to those described:

- This **DDSDomainParticipant** (p. 1139) will attempt to discover peer participants at the given locations but may not succeed if no such participants are available. In this case, this method will not wait for contact attempt(s) to be made and it will not report an error.

- If remote participants described by the given peer description *are* discovered, the distributed application is configured with asymmetric peer lists, and **DDS_DiscoveryQosPolicy::accept_unknown_peers** (p. 585) is set to **DDS_BOOLEAN_TRUE** (p. 298). Thus, this **DDSDomainParticipant** (p. 1139) may actually discover *more* peers than are described in the given peer description.

To be informed of the exact remote participants that are discovered, regardless of which peers this **DDSDomainParticipant** (p. 1139) *attempts* to discover, use the built-in participant topic: **DDS_PARTICIPANT_TOPIC_NAME** (p. 285).

To remove specific peer locators, you may use **DDSDomainParticipant::remove_peer** (p. 1201). If a peer is removed, the add_peer operation will add it back to the list of peers.

To stop communicating with a peer **DDSDomainParticipant** (p. 1139) that has been discovered, use **DDSDomainParticipant::ignore_participant** (p. 1187).

Adding a peer description with this method has no effect on the **DDS_DiscoveryQosPolicy::initial_peers** (p. 583) that may be subsequently retrieved with **DDSDomainParticipant::get_qos()** (p. 1199) (because **DDS_DiscoveryQosPolicy** (p. 582) is immutable).

Parameters:

peer_desc_string *<<in>>* *(p. 200)* New peer descriptor to be added. The format is specified in **Peer Descriptor Format** (p. 389).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

Peer Descriptor Format (p. 389)
DDS_DiscoveryQosPolicy::initial_peers (p. 583)
DDS_PARTICIPANT_TOPIC_NAME (p. 285)
DDSDomainParticipant::get_builtin_subscriber (p. 1186)

6.184.2.67 virtual DDS_ReturnCode_t DDSDomainParticipant::remove_peer (const char * *peer_desc_string*) [pure virtual]

<<eXtension>> (p. 199) Remove one or more peer participants from the list of peers with which this **DDSDomainParticipant** (p. 1139) will try to communicate.

This method may be called any time after this **DDSDomainParticipant** (p. 1139) has been enabled

Calling this method has the following effects:

- If a **DDSDomainParticipant** (p. 1139) was already discovered, it will be locally removed along with all its entities.
- Any further requests coming from a **DDSDomainParticipant** (p. 1139) located on any of the removed peers will be ignored.
- All the locators contained in the peer description will be removed from the peer list. The local **DDSDomainParticipant** (p. 1139) will stop sending announcement to those locators.

If remote participants located on a peer that was previously removed are discovered, they will be ignored untill the related peer is added back by using **DDSDomainParticipant::add_peer** (p. 1199).

Removing a peer description with this method has no effect on the **DDS_DiscoveryQosPolicy::initial_peers** (p. 583) that may be subsequently retrieved with **DDSDomainParticipant::get_qos()** (p. 1199) (because **DDS_DiscoveryQosPolicy** (p. 582) is immutable).

Parameters:

peer_desc_string *<<in>> (p. 200)* Peer descriptor to be removed. The format is specified in **Peer Descriptor Format** (p. 389).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

Peer Descriptor Format (p. 389)
DDS_DiscoveryQosPolicy::initial_peers (p. 583)
DDSDomainParticipant::add_peer (p. 1199)

6.184.2.68 virtual DDS_ReturnCode_t DDSDomainParticipant::set_listener (DDSDomainParticipantListener * *l*, DDS_StatusMask *mask* = DDS_STATUS_MASK_ALL) [pure virtual]

Sets the participant listener.

Parameters:

l <<*in*>> *(p. 200)* Listener to be installed on entity.

mask <<*in*>> *(p. 200)* Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

MT Safety:

Unsafe. This method is not synchronized with the listener callbacks, so it is possible to set a new listener on a participant when the old listener is in a callback.

Care should therefore be taken not to delete any listener that has been set on an enabled participant unless some application-specific means are available of ensuring that the old listener cannot still be in use.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

set_listener (abstract) (p. 1255)

6.184.2.69 virtual DDSDomainParticipantListener* DDSDomainParticipant::get_listener () [pure virtual]

Get the participant listener.

Returns:

Existing listener attached to the **DDSDomainParticipant** (p. 1139).

See also:

get_listener (abstract) (p. 1256)

6.184.2.70 virtual DDSPublisher* DDSDomainParticipant::get_implicit_publisher () [pure virtual]

<<*eXtension*>> *(p. 199)* Returns the implicit **DDSPublisher** (p. 1346). If an implicit Publisher does not already exist, this creates one.

There can only be one implicit Publisher per DomainParticipant.

The implicit Publisher is created with **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) and no Listener.

This implicit Publisher will be deleted automatically when the following methods are called: **DDSDomainParticipant::delete_contained_entities** (p. 1193), or **DDSDomainParticipant::delete_publisher** (p. 1171) with the implicit publisher as a parameter. Additionally, when a DomainParticipant is deleted, if there are no attached DataWriters that belong to the implicit Publisher, the implicit Publisher will be implicitly deleted.

MT Safety:

UNSAFE. It is not safe to create an implicit Publisher while another thread may be simultaneously calling **DDSDomainParticipant::set_default_publisher_qos** (p. 1164).

Returns:

The implicit publisher

See also:

DDS_PUBLISHER_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_publisher (p. 1169)

6.184.2.71 virtual DDSSubscriber* DDSDomainParticipant::get_implicit_subscriber () [pure virtual]

<<*eXtension*>> *(p. 199)* Returns the implicit **DDSSubscriber** (p. 1390). If an implicit Subscriber does not already exist, this creates one.

There can only be one implicit Subscriber per DomainParticipant.

The implicit Subscriber is created with **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) and no Listener.

This implicit Subscriber will be deleted automatically when the following methods are called: **DDSDomainParticipant::delete_contained_entities** (p. 1193), or **DDSDomainParticipant::delete_subscriber** (p. 1173) with the subscriber as a parameter. Additionally, when a DomainParticipant is deleted, if there are no attached DataReaders that belong to the implicit Subscriber, the implicit Subscriber will be implicitly deleted.

MT Safety:

UNSAFE. it is not safe to create the implicit subscriber while another thread may be simultaneously calling **DDSDomainParticipant::set_default_subscriber_qos** (p. 1167).

Returns:

The implicit subscriber

See also:

DDS_PUBLISHER_QOS_DEFAULT (p. 38)
DDSDomainParticipant::create_subscriber (p. 1172)

6.184.2.72 virtual DDSDataWriter* DDSDomainParticipant::create_datawriter (DDSTopic * *topic*, const DDS_DataWriterQos & *qos*, DDSDataWriterListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<*eXtension*>> *(p. 199)* Creates a **DDSDataWriter** (p. 1113) that will be attached and belong to the implicit **DDSPublisher** (p. 1346).

Precondition:

The given **DDSTopic** (p. 1419) must have been created from the same DomainParticipant as the implicit Publisher. If it was created from a different DomainParticipant, this method will fail.

The **DDSDataWriter** (p. 1113) created using this method will be associated with the implicit Publisher. This Publisher is automatically created (if it does not exist) using **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) when the following methods are called: **DDSDomainParticipant::create_datawriter** (p. 1204), **DDSDomainParticipant::create_datawriter_with_profile** (p. 1205), or **DDSDomainParticipant::get_implicit_publisher** (p. 1203).

MT Safety:

UNSAFE. If **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) is used for the `qos` parameter, it is not safe to create the DataWriter while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datawriter_qos** (p. 1150).

Parameters:

topic <<*in*>> *(p. 200)* The **DDSTopic** (p. 1419) that the **DDSDataWriter** (p. 1113) will be associated with.

qos *<<in>> (p. 200)* QoS to be used for creating the new **DDSDataWriter** (p. 1113). The special value **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) can be used to indicate that the **DDSDataWriter** (p. 1113) should be created with the default **DDS_DataWriterQos** (p. 553) set in the implicit **DDSPublisher** (p. 1346). The special value **DDS_DATAWRITER_QOS_USE_TOPIC_QOS** (p. 84) can be used to indicate that the **DDSDataWriter** (p. 1113) should be created with the combination of the default **DDS_DataWriterQos** (p. 553) set on the **DDSPublisher** (p. 1346) and the **DDS_TopicQos** (p. 965) of the **DDSTopic** (p. 1419).

listener *<<in>> (p. 200)* The listener of the **DDSDataWriter** (p. 1113).

mask *<<in>> (p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataWriter** (p. 1113) of a derived class specific to the data type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataWriter (p. 1475)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataWriterQos (p. 553) for rules on consistency among QoS
DDS_DATAWRITER_QOS_DEFAULT (p. 84)
DDS_DATAWRITER_QOS_USE_TOPIC_QOS (p. 84)
DDSDomainParticipant::create_datawriter_with_profile (p. 1205)
DDSDomainParticipant::get_default_datawriter_qos (p. 1150)
DDSDomainParticipant::get_implicit_publisher (p. 1203)
DDSTopic::set_qos (p. 1421)
DDSDataWriter::set_listener (p. 1128)

6.184.2.73 virtual DDSDataWriter* DDSDomainParticipant::create_datawriter_with_profile (DDSTopic * *topic*, const char * *library_name*, const char * *profile_name*, DDSDataWriterListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<eXtension>> (p. 199) Creates a **DDSDataWriter** (p. 1113) using a XML QoS profile that will be attached and belong to the implicit **DDSPublisher** (p. 1346).

Precondition:

The given **DDSTopic** (p. 1419) must have been created from the same DomainParticipant as the implicit Publisher. If it was created from a different DomainParticipant, this method will return NULL.

The **DDSDataWriter** (p. 1113) created using this method will be associated with the implicit Publisher. This Publisher is automatically created (if it does not exist) using **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) when the following methods are called: **DDSDomainParticipant::create_datawriter** (p. 1204), **DDSDomainParticipant::create_datawriter_with_profile** (p. 1205), or **DDSDomainParticipant::get_implicit_publisher** (p. 1203)

Parameters:

topic *<<in>> (p. 200)* The **DDSTopic** (p. 1419) that the **DDSDataWriter** (p. 1113) will be associated with.

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

listener *<<in>> (p. 200)* The listener of the **DDSDataWriter** (p. 1113).

mask *<<in>> (p. 200).* Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataWriter** (p. 1113) of a derived class specific to the data type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataWriter (p. 1475)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataWriterQos (p. 553) for rules on consistency among QoS
DDSDomainParticipant::create_datawriter (p. 1204)
DDSDomainParticipant::get_default_datawriter_qos (p. 1150)
DDSDomainParticipant::get_implicit_publisher (p. 1203)
DDSTopic::set_qos (p. 1421)
DDSDataWriter::set_listener (p. 1128)

6.184.2.74 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_datawriter (DDSDataWriter * *a_datawriter*) [pure virtual]

<<*eXtension*>> *(p. 199)* Deletes a **DDSDataWriter** (p. 1113) that belongs to the implicit **DDSPublisher** (p. 1346).

The deletion of the **DDSDataWriter** (p. 1113) will automatically unregister all instances. Depending on the settings of the **WRITER_DATA_LIFECYCLE** (p. 375) QosPolicy, the deletion of the **DDSDataWriter** (p. 1113) may also dispose all instances.

6.184.3 Special Instructions if Using 'Timestamp' APIs and BY_SOURCE_TIMESTAMP Destination Ordering:

If the DataWriter's **DDS_DestinationOrderQosPolicy::kind** (p. 572) is **DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS** (p. 366), calls to **delete_datawriter()** (p. 1207) may fail if your application has previously used the 'with timestamp' APIs (write_w_timestamp(), register_instance_w_timestamp(), unregister_instance_w_timestamp(), or dispose_w_timestamp()) with a timestamp larger (later) than the time at which **delete_datawriter()** (p. 1207) is called. To prevent **delete_datawriter()** (p. 1207) from failing in this situation, either:

- Change the **WRITER_DATA_LIFECYCLE** (p. 375) QosPolicy so that RTI Connext will not autodispose unregistered instances (set **DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances** (p. 1072) to **DDS_BOOLEAN_FALSE** (p. 299).) or

- Explicitly call unregister_instance_w_timestamp() for all instances modified with the *_w_timestamp() APIs before calling **delete_datawriter()** (p. 1207).

Precondition:

If the **DDSDataWriter** (p. 1113) does not belong to the implicit **DDSPublisher** (p. 1346), the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSDataWriter** (p. 1113) will not be called after this method completes successfully.

Parameters:

a_datawriter *<<in>>* *(p. 200)* The **DDSDataWriter** (p. 1113) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

See also:

DDSDomainParticipant::get_implicit_publisher (p. 1203)

6.184.3.1 virtual DDSDataReader* DDSDomainParticipant::create_datareader (DDSTopicDescription * *topic*, const DDS_DataReaderQos & *qos*, DDSDataReaderListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<eXtension>> *(p. 199)* Creates a **DDSDataReader** (p. 1087) that will be attached and belong to the implicit **DDSSubscriber** (p. 1390).

Precondition:

The given **DDSTopicDescription** (p. 1427) must have been created from the same DomainParticipant as the implicit Subscriber. If it was created from a different DomainParticipant, this method will return NULL.

The **DDSDataReader** (p. 1087) created using this method will be associated with the implicit Subscriber. This Subscriber is automatically created (if it does not exist) using **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) when the following methods are called: **DDSDomainParticipant::create_datareader** (p. 1208), **DDSDomainParticipant::create_datareader_with_profile** (p. 1209), or **DDSDomainParticipant::get_implicit_subscriber** (p. 1203).

MT Safety:

UNSAFE. If **DDS_DATAREADER_QOS_DEFAULT** (p. 99) is used for the `qos` parameter, it is not safe to create the datareader while another thread may be simultaneously calling **DDSDomainParticipant::set_default_datareader_qos** (p. 1153).

Parameters:

topic *<<in>>* *(p. 200)* The **DDSTopicDescription** (p. 1427) that the **DDSDataReader** (p. 1087) will be associated with.

qos *<<in>> (p. 200)* The qos of the **DDSDataReader** (p. 1087). The special value **DDS_DATAREADER_QOS_DEFAULT** (p. 99) can be used to indicate that the **DDSDataReader** (p. 1087) should be created with the default **DDS_DataReaderQos** (p. 515) set in the implicit **DDSSubscriber** (p. 1390). If **DDSTopicDescription** (p. 1427) is of type **DDSTopic** (p. 1419) or **DDSContentFilteredTopic** (p. 1081), the special value **DDS_DATAREADER_QOS_USE_TOPIC_QOS** (p. 99) can be used to indicate that the **DDSDataReader** (p. 1087) should be created with the combination of the default **DDS_DataReaderQos** (p. 515) set on the implicit **DDSSubscriber** (p. 1390) and the **DDS_TopicQos** (p. 965) (in the case of a **DDSContentFilteredTopic** (p. 1081), the **DDS_TopicQos** (p. 965) of the related **DDSTopic** (p. 1419)). if **DDS_DATAREADER_QOS_USE_TOPIC_QOS** (p. 99) is used, `topic` cannot be a **DDSMultiTopic** (p. 1322).

listener *<<in>> (p. 200)* The listener of the **DDSDataReader** (p. 1087).

mask *<<in>> (p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataReader** (p. 1087) of a derived class specific to the data-type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataReader (p. 1444)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataReaderQos (p. 515) for rules on consistency among QoS
DDSDomainParticipant::create_datareader_with_profile (p. 1209)
DDSDomainParticipant::get_default_datareader_qos (p. 1152)
DDSDomainParticipant::get_implicit_subscriber (p. 1203)
DDSTopic::set_qos (p. 1421)
DDSDataReader::set_listener (p. 1104)

6.184.3.2 virtual DDSDataReader* DDSDomainParticipant::create_datareader_with_profile (DDSTopicDescription * *topic*, const char * *library_name*, const char * *profile_name*, DDSDataReaderListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<eXtension>> (p. 199) Creates a **DDSDataReader** (p. 1087) using a XML QoS profile that will be attached and belong to the implicit **DDSSubscriber** (p. 1390).

Precondition:

The given **DDSTopicDescription** (p. 1427) must have been created from the same DomainParticipant as the implicit subscriber. If it was created from a different DomainParticipant, this method will return NULL.

The **DDSDataReader** (p. 1087) created using this method will be associated with the implicit Subscriber. This Subscriber is automatically created (if it does not exist) using **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) when the following methods are called: **DDSDomainParticipant::create_datareader** (p. 1208), **DDSDomainParticipant::create_datareader_with_profile** (p. 1209), or **DDSDomainParticipant::get_implicit_subscriber** (p. 1203)

Parameters:

topic *<<in>> (p. 200)* The **DDSTopicDescription** (p. 1427) that the **DDSDataReader** (p. 1087) will be associated with.

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

listener *<<in>> (p. 200)* The listener of the **DDSDataReader** (p. 1087).

mask *<<in>> (p. 200).* Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataReader** (p. 1087) of a derived class specific to the data-type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataReader (p. 1444)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataReaderQos (p. 515) for rules on consistency among QoS
DDSDomainParticipant::create_datareader (p. 1208)
DDSDomainParticipant::get_default_datareader_qos (p. 1152)
DDSDomainParticipant::get_implicit_subscriber (p. 1203)
DDSTopic::set_qos (p. 1421)
DDSDataReader::set_listener (p. 1104)

6.184.3.3 virtual DDS_ReturnCode_t DDSDomainParticipant::delete_datareader (DDSDataReader * *a_datareader*) [pure virtual]

<<eXtension>> (p. 199) Deletes a **DDSDataReader** (p. 1087) that belongs to the implicit **DDSSubscriber** (p. 1390).

Precondition:

If the **DDSDataReader** (p. 1087) does not belong to the implicit **DDSSubscriber** (p. 1390), or if ther are any existing **DDSReadCondition** (p. 1374) or **DDSQueryCondition** (p. 1372) objects that are attached to the **DDSDataReader** (p. 1087), or if there are outstanding loans on samples (as a result of a call to read(), take(), or one of the variants thereof), the operation fails with the error **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSDataReader** (p. 1087) will not be called after this method completes successfully.

Parameters:

a_datareader *<<in>> (p. 200)* The **DDSDataReader** (p. 1087) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

See also:

DDSDomainParticipant::get_implicit_subscriber (p. 1203)

6.184.3.4 virtual DDSPublisher* DDSDomainParticipant::lookup_publisher_by_name_exp (const char * *publisher_name*) [pure virtual]

<<experimental>> (p. 199) <<eXtension>> (p. 199) Looks up a **DDSPublisher** (p. 1346) by its entity name within this **DDSDomainParticipant** (p. 1139).

Every **DDSPublisher** (p. 1346) in the system has an entity name which is configured and stored in the GROUP_DATA policy. The use of the GROUP_DATA to store the entity name is a temporary situation while the feature is in experimental state.

This operation retrieves a **DDSPublisher** (p. 1346) within the **DDSDomainParticipant** (p. 1139) given the entity's name. If there are several **DDSPublisher** (p. 1346) with the same name within the **DDSDomainParticipant** (p. 1139), this function returns the first matching occurrence.

Parameters:

publisher_name *<<in>> (p. 200)* Entity name of the **DDSPublisher** (p. 1346).

Returns:

The first **DDSPublisher** (p. 1346) found with the specified name or NULL if it is not found.

See also:

DDSDomainParticipant::lookup_datawriter_by_name_exp (p. 1213)

6.184.3.5 virtual DDSSubscriber* DDSDomainParticipant::lookup_subscriber_by_name_exp (const char * *subscriber_name*) [pure virtual]

<<experimental>> (p. 199) <<eXtension>> (p. 199) Retrieves a **DDSSubscriber** (p. 1390) by its entity name within this **DDSDomainParticipant** (p. 1139).

Every **DDSSubscriber** (p. 1390) in the system has an entity name which is configured and stored in the GROUP_DATA policy. The use of the GROUP_DATA to store the entity name is a temporary situation while the function is in experimental state.

This operation retrieves a **DDSSubscriber** (p. 1390) within the **DDSDomainParticipant** (p. 1139) given the entity's name. If there are several **DDSSubscriber** (p. 1390) with the same name within the **DDSDomainParticipant** (p. 1139), this function returns the first matching occurrence.

Parameters:

subscriber_name *<<in>> (p. 200)* Entity name of the **DDSSubscriber** (p. 1390).

Returns:

The first **DDSSubscriber** (p. 1390) found with the specified name or NULL if it is not found.

See also:

DDSDomainParticipant::lookup_datareader_by_name_exp (p. 1214)

6.184.3.6 virtual DDSDataWriter* DDSDomainParticipant::lookup_datawriter_by_name_exp (const char * *datawriter_full_name*) [pure virtual]

<<experimental>> (p. *199*) *<<eXtension>>* (p. *199*) Looks up a **DDSDataWriter** (p. 1113) by its entity name within this **DDSDomainParticipant** (p. 1139).

Every **DDSDataWriter** (p. 1113) in the system has an entity name which is configured and stored in the EntityName policy, **ENTITY_NAME** (p. 445).

Every **DDSPublisher** (p. 1346) in the system has an entity name which is configured and stored in the *<<eXtension>>* (p. *199*) GROUP_DATA policy. This is a temporary situation situation while the function is in experimental state.

This operation retrieves a **DDSDataWriter** (p. 1113) within a **DDSPublisher** (p. 1346) given the specified name which encodes both to the **DDSDataWriter** (p. 1113) and the **DDSPublisher** (p. 1346) name.

The specified name might be given as a fully-qualified entity name or as a plain name.

The fully qualified entity name is a concatenation of the **DDSPublisher** (p. 1346) to which the **DDSDataWriter** (p. 1113) belongs and the entity name of of the **DDSDataWriter** (p. 1113) itself, separated by a double colon "::". For example: MyPublisherName::MyDataWriterName

The plain name contains the **DDSDataWriter** (p. 1113) name only. In this situation it is implied that the **DDSDataWriter** (p. 1113) belongs to the implicit **DDSPublisher** (p. 1346) so the use of a plain name is equivalent to specifying a fully qualified name with the **DDSPublisher** (p. 1346) name part being "implicit". For example: the plain name "MyDataWriterName" is equivalent to specifiying the fully qualified name "implicit::MyDataWriterName"

The **DDSDataWriter** (p. 1113) is only looked up within the **DDSPublisher** (p. 1346) specified in the fully qualified name, or within the implicit **DDSPublisher** (p. 1346) if the name was not fully qualified.

If there are several **DDSDataWriter** (p. 1113) with the same name within the corresponding **DDSPublisher** (p. 1346) this function returns the first matching occurrence.

Parameters:

datawriter_full_name *<<**in**>>* *(p. 200)* Entity name or fully-qualified entity name of the **DDSDataWriter** (p. 1113).

Returns:

The first **DDSDataWriter** (p. 1113) found with the specified name or NULL if it is not found.

See also:

DDSPublisher::lookup_datawriter_by_name_exp (p. 1368)
DDSDomainParticipant::lookup_publisher_by_name_exp (p. 1211)

6.184.3.7 virtual DDSDataReader* DDSDomainParticipant::lookup_datareader_by_name_exp (const char * *datareader_full_name*) [pure virtual]

*<<**experimental**>>* *(p. 199)* *<<**eXtension**>>* *(p. 199)* Retrieves up a **DDSDataReader** (p. 1087) by its entity name in this **DDSDomainParticipant** (p. 1139).

Every **DDSDataReader** (p. 1087) in the system has an entity name which is configured and stored in the EntityName policy, **ENTITY_NAME** (p. 445).

Every **DDSSubscriber** (p. 1390) in the system has an entity name which is configured and stored in the *<<**eXtension**>>* *(p. 199)* GROUP_DATA policy. This is a temporary situation situation while the function is in experimental state.

This operation retrieves a **DDSDataReader** (p. 1087) within a **DDSSubscriber** (p. 1390) given the specified name which encodes both to the **DDSDataReader** (p. 1087) and the **DDSSubscriber** (p. 1390) name.

The specified name might be given as a fully-qualified entity name or as a plain name.

The fully qualified entity name is a concatenation of the **DDSSubscriber** (p. 1390) to which the **DDSDataReader** (p. 1087) belongs and the entity name of of the **DDSDataReader** (p. 1087) itself, separated by a double colon "::". For example: MySubscriberName::MyDataReaderName

The plain name contains the **DDSDataReader** (p. 1087) name only. In this situation it is implied that the **DDSDataReader** (p. 1087) belongs to the implicit **DDSSubscriber** (p. 1390) so the use of a plain name is equivalent to specifying a fully qualified name with the **DDSSubscriber** (p. 1390) name part being "implicit". For example: the plain name "MyDataReaderName" is equivalent to specifiying the fully qualified name "implicit::MyDataReaderName"

The **DDSDataReader** (p. 1087) is only looked up within the **DDSSubscriber** (p. 1390) specified in the fully qualified name, or within the implicit **DDSSubscriber** (p. 1390) if the name was not fully qualified.

If there are several **DDSDataReader** (p. 1087) with the same name within the corresponding **DDSSubscriber** (p. 1390) this function returns the first matching occurrence.

Parameters:

datareader_full_name *<<in>> (p. 200)* Full entity name of the **DDSDataReader** (p. 1087).

Returns:

The first **DDSDataReader** (p. 1087) found with the specified name or NULL if it is not found.

See also:

DDSSubscriber::lookup_datareader_by_name_exp (p. 1412)
DDSDomainParticipant::lookup_subscriber_by_name_exp (p. 1212)

6.185 DDSDomainParticipantFactory Class Reference

*<<**singleton**>> (p. 200) <<**interface**>> (p. 199)* Allows creation and destruction of **DDSDomainParticipant** (p. 1139) objects.

Public Member Functions

- virtual **DDS_ReturnCode_t set_default_participant_qos** (const **DDS_DomainParticipantQos** &qos)=0

  *Sets the default **DDS_DomainParticipantQos** (p. 588) values for this domain participant factory.*

- virtual **DDS_ReturnCode_t set_default_participant_qos_with_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Sets the default **DDS_DomainParticipantQos** (p. 588) values for this domain participant factory based on the input XML QoS profile.*

- virtual **DDS_ReturnCode_t get_default_participant_qos** (**DDS_DomainParticipantQos** &qos)=0

  *Initializes the **DDS_DomainParticipantQos** (p. 588) instance with default values.*

- virtual **DDS_ReturnCode_t set_default_library** (const char *library_name)=0

  <<**eXtension**>> (p. 199) *Sets the default XML library for a **DDSDomainParticipantFactory** (p. 1216).*

- virtual const char * **get_default_library** ()=0

  <<**eXtension**>> (p. 199) *Gets the default XML library associated with a **DDSDomainParticipantFactory** (p. 1216).*

- virtual **DDS_ReturnCode_t set_default_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Sets the default XML profile for a **DDSDomainParticipantFactory** (p. 1216).*

- virtual const char * **get_default_profile** ()=0

  <<**eXtension**>> (p. 199) *Gets the default XML profile associated with a **DDSDomainParticipantFactory** (p. 1216).*

- virtual const char * **get_default_profile_library** ()=0

<<**eXtension**>> (p. 199) *Gets the library where the default XML profile is contained for a* ***DDSDomainParticipantFactory*** *(p. 1216).*

ˆ virtual **DDS_ReturnCode_t get_participant_qos_from_profile** (**DDS_DomainParticipantQos** &qos, const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_DomainParticipantQos*** *(p. 588) values associated with the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t get_publisher_qos_from_profile** (**DDS_PublisherQos** &qos, const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_PublisherQos*** *(p. 851) values associated with the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t get_subscriber_qos_from_profile** (**DDS_SubscriberQos** &qos, const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_SubscriberQos*** *(p. 934) values associated with the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t get_datawriter_qos_from_profile** (**DDS_DataWriterQos** &qos, const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_DataWriterQos*** *(p. 553) values associated with the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t get_datawriter_qos_from_profile_w_topic_name** (**DDS_DataWriterQos** &qos, const char *library_name, const char *profile_name, const char *topic_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_DataWriterQos*** *(p. 553) values associated with the input XML QoS profile while applying topic filters to the input topic name.*

ˆ virtual **DDS_ReturnCode_t get_datareader_qos_from_profile** (**DDS_DataReaderQos** &qos, const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_DataReaderQos*** *(p. 515) values associated with the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t get_datareader_qos_from_profile_w_topic_name** (**DDS_DataReaderQos** &qos, const char *library_name, const char *profile_name, const char *topic_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_DataReaderQos*** *(p. 515) values associated with the input XML QoS profile while applying topic filters to the input topic name.*

^ virtual **DDS_ReturnCode_t get_topic_qos_from_profile** (**DDS_TopicQos** &qos, const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_TopicQos*** *(p. 965) values associated with the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t get_topic_qos_from_profile_w_topic_name** (**DDS_TopicQos** &qos, const char *library_name, const char *profile_name, const char *topic_name)=0

<<**eXtension**>> (p. 199) *Gets the* ***DDS_TopicQos*** *(p. 965) values associated with the input XML QoS profile while applying topic filters to the input topic name.*

^ virtual **DDS_ReturnCode_t get_qos_profile_libraries** (struct **DDS_StringSeq** &library_names)=0

<<**eXtension**>> (p. 199) *Gets the names of all XML QoS profile libraries associated with the* ***DDSDomainParticipantFactory*** *(p. 1216)*

^ virtual **DDS_ReturnCode_t get_qos_profiles** (struct **DDS_StringSeq** &profile_names, const char *library_name)=0

<<**eXtension**>> (p. 199) *Gets the names of all XML QoS profiles associated with the input XML QoS profile library.*

^ virtual **DDSDomainParticipant** * **create_participant** (**DDS_DomainId_t** domainId, const **DDS_DomainParticipantQos** &qos, **DDSDomainParticipantListener** *listener, **DDS_StatusMask** mask)=0

Creates a new ***DDSDomainParticipant*** *(p. 1139) object.*

^ virtual **DDSDomainParticipant** * **create_participant_with_profile** (**DDS_DomainId_t** domainId, const char *library_name, const char *profile_name, **DDSDomainParticipantListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a new* ***DDSDomainParticipant*** *(p. 1139) object using the* ***DDS_DomainParticipantQos*** *(p. 588) associated with the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t delete_participant** (**DDSDomainParticipant** *a_participant)=0

Deletes an existing ***DDSDomainParticipant*** *(p. 1139).*

^ virtual **DDSDomainParticipant** * **lookup_participant** (**DDS_DomainId_t** domainId)=0

Locates an existing ***DDSDomainParticipant*** *(p. 1139).*

^ virtual **DDS_ReturnCode_t set_qos** (const **DDS_DomainParticipantFactoryQos** &qos)=0

Sets the value for a participant factory QoS.

^ virtual **DDS_ReturnCode_t get_qos** (**DDS_DomainParticipantFactoryQos** &qos)=0

Gets the value for participant factory QoS.

^ virtual **DDS_ReturnCode_t load_profiles** ()=0

<<**eXtension**>> (p. 199) *Loads the XML QoS profiles.*

^ virtual **DDS_ReturnCode_t reload_profiles** ()=0

<<**eXtension**>> (p. 199) *Reloads the XML QoS profiles.*

^ virtual **DDS_ReturnCode_t unload_profiles** ()=0

<<**eXtension**>> (p. 199) *Unloads the XML QoS profiles.*

^ virtual **DDS_ReturnCode_t unregister_thread** ()=0

<<**eXtension**>> (p. 199) *Allows the user to release thread specific resources kept by the middleware.*

^ virtual **DDSDomainParticipant** * **create_participant_from_config_exp** (const char *configuration_name, const char *participant_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Creates a **DDSDomainParticipant** (p. 1139) given its configuration name from a description provided in an XML configuration file.*

^ virtual **DDSDomainParticipant** * **lookup_participant_by_name_exp** (const char *participant_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Looks up a **DDSDomainParticipant** (p. 1139) by its entity name in the **DDSDomainParticipantFactory** (p. 1216).*

^ virtual **DDS_ReturnCode_t register_type_support_exp** (**DDSDomainParticipantFactory_RegisterTypeFunction** register_type_fcn, const char *type_name)=0

<<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Registers a **DDSTypeSupport** (p. 1432) with the **DDSDomainParticipantFactory** (p. 1216) to enable automatic registration if the corresponding type, should it be needed by a **DDSDomainParticipant** (p. 1139).*

Static Public Member Functions

- static **DDSDomainParticipantFactory** * **get_instance** ()
  Gets the singleton instance of this class.

- static **DDS_ReturnCode_t finalize_instance** ()
  <<**eXtension**>> (p. 199) *Destroys the singleton instance of this class.*

6.185.1 Detailed Description

<<***singleton***>> *(p. 200)* <<***interface***>> *(p. 199)* Allows creation and destruction of **DDSDomainParticipant** (p. 1139) objects.

The sole purpose of this class is to allow the creation and destruction of **DDSDomainParticipant** (p. 1139) objects. This class itself is a <<***singleton***>> *(p. 200)*, and accessed via the **get_instance()** (p. 1221) method, and destroyed with **finalize_instance()** (p. 1221) method.

A single application can participate in multiple domains by instantiating multiple **DDSDomainParticipant** (p. 1139) objects.

An application may even instantiate multiple participants in the same domain. Participants in the same domain exchange data in the same way regardless of whether they are in the same application or different applications or on the same node or different nodes; their location is transparent.

There are two important caveats:

- When there are multiple participants on the same node (in the same application or different applications) in the same domain, the application(s) must make sure that the participants do not try to bind to the same port numbers. You must disambiguate between the participants by setting a participant ID for each participant (**DDS_WireProtocolQosPolicy::participant_id** (p. 1063)). The port numbers used by a participant are calculated based on both the participant index and the domain ID, so if all participants on the same node have different participant indexes, they can coexist in the same domain.

- You cannot mix entities from different participants. For example, you cannot delete a topic on a different participant than you created it from, and you cannot ask a subscriber to create a reader for a topic created from a participant different than the subscriber's own participant. (Note that it is permissable for an application built on top of RTI Connext to know about entities from different participants. For example, an application could keep references to a reader from one domain and a writer from another and then bridge the domains by writing the data received in the reader callback.)

See also:

DDSDomainParticipant (p. 1139)

6.185.2 Member Function Documentation

6.185.2.1 static DDSDomainParticipantFactory* DDSDomainParticipantFactory::get_instance () [static]

Gets the singleton instance of this class.

MT Safety:

On non-Linux systems: UNSAFE for multiple threads to simultaneously make the FIRST call to either **DDSDomainParticipantFactory::get_instance()** (p. 1221) or **DDSDomainParticipantFactory::finalize_instance()** (p. 1221). Subsequent calls are thread safe. (On Linux systems, these calls are thread safe.)

DDS_TheParticipantFactory can be used as an alias for the singleton factory returned by this operation.

Returns:

The singleton **DDSDomainParticipantFactory** (p. 1216) instance.

See also:

DDS_TheParticipantFactory

6.185.2.2 static DDS_ReturnCode_t DDSDomainParticipantFactory::finalize_instance () [static]

<<***eXtension***>> *(p. 199)* Destroys the singleton instance of this class.

Only necessary to explicitly reclaim resources used by the participant factory singleton. Note that on many OSs, these resources are automatically reclaimed by the OS when the program terminates. However, some memory-check tools still flag these as unreclaimed. So this method provides a way to clean up memory used by the participant factory.

Precondition:

All participants created from the factory have been deleted.

Postcondition:

All resources belonging to the factory have been reclaimed. Another call to **DDSDomainParticipantFactory::get_instance** (p. 1221) will return a new lifecycle of the singleton.

MT Safety:

On non-Linux systems: UNSAFE for multiple threads to simultaneously make the FIRST call to either **DDSDomainParticipantFactory::get_instance()** (p. 1221) or **DDSDomainParticipantFactory::finalize_instance()** (p. 1221). Subsequent calls are thread safe. (On Linux systems, these calls are thread safe.)

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

6.185.2.3 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::set_default_participant_qos (const DDS_DomainParticipantQos & *qos*) [pure virtual]

Sets the default **DDS_DomainParticipantQos** (p. 588) values for this domain participant factory.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

MT Safety:

UNSAFE. It is not safe to retrieve the default QoS value from a domain participant factory while another thread may be simultaneously calling **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222)

Parameters:

qos <<***inout***>> *(p. 200)* Qos to be filled up. The special value **DDS_PARTICIPANT_QOS_DEFAULT** (p. 35) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_PARTICIPANT_QOS_DEFAULT (p. 35)
DDSDomainParticipantFactory::create_participant (p. 1233)

6.185.2.4 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::set_default_participant_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> (p. 199) Sets the default **DDS_DomainParticipantQos** (p. 588) values for this domain participant factory based on the input XML QoS profile.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

This default value will be used for newly created **DDSDomainParticipant** (p. 1139) if **DDS_PARTICIPANT_QOS_DEFAULT** (p. 35) is specified as the `qos` parameter when **DDSDomainParticipantFactory::create_participant** (p. 1233) is called.

Precondition:

The **DDS_DomainParticipantQos** (p. 588) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to retrieve the default QoS value from a domain participant factory while another thread may be simultaneously calling **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222)

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_PARTICIPANT_QOS_DEFAULT (p. 35)
DDSDomainParticipantFactory::create_participant_with_profile (p. 1235)

6.185.2.5 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_default_participant_qos (DDS_DomainParticipantQos & *qos*) [pure virtual]

Initializes the **DDS_DomainParticipantQos** (p. 588) instance with default values.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222), or **DDSDomainParticipantFactory::set_default_participant_qos_with_profile** (p. 1223), or else, if the call was never made, the default values listed in **DDS_DomainParticipantQos** (p. 588).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos *<<**out**>>* *(p. 200)* the domain participant's QoS

MT Safety:

UNSAFE. It is not safe to retrieve the default QoS value from a domain participant factory while another thread may be simultaneously calling **DDSDomainParticipantFactory::set_default_participant_qos** (p. 1222)

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_PARTICIPANT_QOS_DEFAULT (p. 35)
DDSDomainParticipantFactory::create_participant (p. 1233)

6.185.2.6 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::set_default_library (const char * *library_name*) [pure virtual]

<<eXtension>> (p. 199) Sets the default XML library for a **DDSDomainParticipantFactory** (p. 1216).

Any API requiring a library_name as a parameter can use null to refer to the default library.

See also:

DDSDomainParticipantFactory::set_default_profile (p. 1225) for more information.

Parameters:

library_name *<<in>> (p. 200)* Library name. If library_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDomainParticipantFactory::get_default_library (p. 1225)

6.185.2.7 virtual const char* DDSDomainParticipantFactory::get_default_library () [pure virtual]

<<eXtension>> (p. 199) Gets the default XML library associated with a **DDSDomainParticipantFactory** (p. 1216).

Returns:

The default library or null if the default library was not set.

See also:

DDSDomainParticipantFactory::set_default_library (p. 1225)

6.185.2.8 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::set_default_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> (p. 199) Sets the default XML profile for a **DDSDomainParticipantFactory** (p. 1216).

This method specifies the profile that will be used as the default the next time a default DomainParticipantFactory profile is needed during a call to a DomainParticipantFactory method. When calling a **DDSDomainParticipantFactory** (p. 1216) method that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

This method does not set the default QoS for newly created DomainParticipants; for this functionality, use **DDSDomainParticipantFactory::set_default_participant_qos_with_profile** (p. 1223) (you may pass in NULL after having called **set_default_profile()** (p. 1225)).

Parameters:

library_name *<<in>> (p. 200)* The library name containing the profile.

profile_name *<<in>> (p. 200)* The profile name. If profile_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSDomainParticipantFactory::get_default_profile (p. 1226)
DDSDomainParticipantFactory::get_default_profile_library (p. 1226)

6.185.2.9 virtual const char* DDSDomainParticipantFactory::get_default_profile () [pure virtual]

<<eXtension>> (p. 199) Gets the default XML profile associated with a **DDSDomainParticipantFactory** (p. 1216).

Returns:

The default profile or null if the default profile was not set.

See also:

DDSDomainParticipantFactory::set_default_profile (p. 1225)

6.185.2.10 virtual const char* DDSDomainParticipantFactory::get_default_profile_library () [pure virtual]

<<eXtension>> (p. 199) Gets the library where the default XML profile is contained for a **DDSDomainParticipantFactory** (p. 1216).

The default profile library is automatically set when **DDSDomainParticipantFactory::set_default_profile** (p. 1225) is called.

This library can be different than the **DDSDomainParticipantFactory** (p. 1216) default library (see **DDSDomainParticipantFactory::get_default_library** (p. 1225)).

Returns:

The default profile library or null if the default profile was not set.

See also:

DDSDomainParticipantFactory::set_default_profile (p. 1225)

6.185.2.11 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_participant_qos_from_profile (DDS_DomainParticipantQos & *qos*, const char * *library_name*, const char * *profile_name*) [pure virtual]

<<*eXtension*>> *(p. 199)* Gets the **DDS_DomainParticipantQos** (p. 588) values associated with the input XML QoS profile.

Parameters:

qos <<*out*>> *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name <<*in*>> *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name <<*in*>> *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.12 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_publisher_qos_from_profile (DDS_PublisherQos & *qos*, const char * *library_name*, const char * *profile_name*) [pure virtual]

<<*eXtension*>> (p. 199) Gets the **DDS_PublisherQos** (p. 851) values associated with the input XML QoS profile.

Parameters:

qos <<*out*>> (p. 200) Qos to be filled up. Cannot be NULL.

library_name <<*in*>> (p. 200) Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name <<*in*>> (p. 200) XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.13 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_subscriber_qos_from_profile (DDS_SubscriberQos & *qos*, const char * *library_name*, const char * *profile_name*) [pure virtual]

<<*eXtension*>> (p. 199) Gets the **DDS_SubscriberQos** (p. 934) values associated with the input XML QoS profile.

Parameters:

qos <<*out*>> (p. 200) Qos to be filled up. Cannot be NULL.

library_name <<*in*>> (p. 200) Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name <<*in*>> (p. 200) XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_-ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.14 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_datawriter_qos_from_profile (DDS_DataWriterQos & *qos*, const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Gets the **DDS_DataWriterQos** (p. 553) values associated with the input XML QoS profile.

Parameters:

qos *<<out>>* *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_-ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.15 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_datawriter_qos_from_profile_w_topic_name (DDS_DataWriterQos & *qos*, const char * *library_name*, const char * *profile_name*, const char * *topic_name*) [pure virtual]

<<eXtension>> *(p. 199)* Gets the **DDS_DataWriterQos** (p. 553) values associated with the input XML QoS profile while applying topic filters to the input topic name.

Parameters:

qos *<<**out**>>* *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name *<<**in**>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<**in**>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

topic_name *<<**in**>>* *(p. 200)* Topic name that will be evaluated against the topic_filter attribute in the XML QoS profile. If topic_name is null, RTI Connext will match only QoSs without explicit topic_filter expressions.

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.16 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_datareader_qos_from_profile (DDS_DataReaderQos & *qos*, const char * *library_name*, const char * *profile_name*) [pure virtual]

*<<**eXtension**>>* *(p. 199)* Gets the **DDS_DataReaderQos** (p. 515) values associated with the input XML QoS profile.

Parameters:

qos *<<**out**>>* *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name *<<**in**>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<**in**>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.17 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_datareader_qos_from_profile_w_topic_name (DDS_DataReaderQos & *qos*, const char * *library_name*, const char * *profile_name*, const char * *topic_name*) [pure virtual]

<<eXtension>> *(p. 199)* Gets the **DDS_DataReaderQos** (p. 515) values associated with the input XML QoS profile while applying topic filters to the input topic name.

Parameters:

qos *<<out>>* *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

topic_name *<<in>>* *(p. 200)* Topic name that will be evaluated against the topic_filter attribute in the XML QoS profile. If topic_name is null, RTI Connext will match only QoSs without explicit topic_filter expressions.

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.18 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_topic_qos_from_profile (DDS_TopicQos & *qos*, const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Gets the **DDS_TopicQos** (p. 965) values associated with the input XML QoS profile.

Parameters:

qos *<<**out**>>* *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name *<<**in**>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<**in**>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.19 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_topic_qos_from_profile_w_topic_name (DDS_TopicQos & *qos*, const char * *library_name*, const char * *profile_name*, const char * *topic_name*) [pure virtual]

*<<**eXtension**>>* *(p. 199)* Gets the **DDS_TopicQos** (p. 965) values associated with the input XML QoS profile while applying topic filters to the input topic name.

Parameters:

qos *<<**out**>>* *(p. 200)* Qos to be filled up. Cannot be NULL.

library_name *<<**in**>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name *<<**in**>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

topic_name *<<**in**>>* *(p. 200)* Topic name that will be evaluated against the topic_filter attribute in the XML QoS profile. If topic_name is null, RTI Connext will match only QoSs without explicit topic_filter expressions.

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.20 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_qos_profile_libraries (struct DDS_StringSeq & *library_names*) [pure virtual]

<<eXtension>> (p. 199) Gets the names of all XML QoS profile libraries associated with the **DDSDomainParticipantFactory** (p. 1216)

Parameters:

library_names *<<out>>* (p. 200) **DDS_StringSeq** (p. 929) to be filled with names of XML QoS profile libraries. Cannot be NULL.

6.185.2.21 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_qos_profiles (struct DDS_StringSeq & *profile_names*, const char * *library_name*) [pure virtual]

<<eXtension>> (p. 199) Gets the names of all XML QoS profiles associated with the input XML QoS profile library.

Parameters:

profile_names *<<out>>* (p. 200) **DDS_StringSeq** (p. 929) to be filled with names of XML QoS profiles. Cannot be NULL.

library_name *<<in>>* (p. 200) Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

6.185.2.22 virtual DDSDomainParticipant* DDSDomainParticipantFactory::create_participant (DDS_DomainId_t *domainId*, const DDS_DomainParticipantQos & *qos*, DDSDomainParticipantListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

Creates a new **DDSDomainParticipant** (p. 1139) object.

Precondition:

The specified QoS policies must be consistent or the operation will fail and no **DDSDomainParticipant** (p. 1139) will be created.

If you want to create multiple participants on a given host in the same domain, make sure each one has a different participant index (set in the **DDS_WireProtocolQosPolicy** (p. 1059)). This in turn will ensure each participant uses a different port number (since the unicast port numbers are calculated from the participant index and the domain ID).

Note that if there is a single participant per host in a given domain, the participant index can be left at the default value (-1).

Parameters:

domainId *<<in>> (p. 200)* ID of the domain that the application intends to join. **[range]** [>=0], and does not violate guidelines stated in **DDS_RtpsWellKnownPorts_t** (p. 905).

qos *<<in>> (p. 200)* the DomainParticipant's QoS. The special value **DDS_PARTICIPANT_QOS_DEFAULT** (p. 35) can be used to indicate that the **DDSDomainParticipant** (p. 1139) should be created with the default **DDS_DomainParticipantQos** (p. 588) set in the **DDSDomainParticipantFactory** (p. 1216).

listener *<<in>> (p. 200)* the domain participant's listener.

mask *<<in>> (p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

domain participant or NULL on failure

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DomainParticipantQos (p. 588) for rules on consistency among QoS
DDS_PARTICIPANT_QOS_DEFAULT (p. 35)
NDDS_DISCOVERY_PEERS (p. 388)
DDSDomainParticipantFactory::create_participant_with_profile() (p. 1235)
DDSDomainParticipantFactory::get_default_participant_qos() (p. 1224)
DDSDomainParticipant::set_listener() (p. 1202)

6.185.2.23 virtual DDSDomainParticipant* DDSDomainParticipantFactory::create_participant_with_profile (DDS_DomainId_t *domainId*, const char * *library_name*, const char * *profile_name*, DDSDomainParticipantListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<*eXtension*>> *(p. 199)* Creates a new **DDSDomainParticipant** (p. 1139) object using the **DDS_DomainParticipantQos** (p. 588) associated with the input XML QoS profile.

Precondition:

The **DDS_DomainParticipantQos** (p. 588) in the input profile must be consistent, or the operation will fail and no **DDSDomainParticipant** (p. 1139) will be created.

If you want to create multiple participants on a given host in the same domain, make sure each one has a different participant index (set in the **DDS_WireProtocolQosPolicy** (p. 1059)). This in turn will ensure each participant uses a different port number (since the unicast port numbers are calculated from the participant index and the domain ID).

Note that if there is a single participant per host in a given domain, the participant index can be left at the default value (-1).

Parameters:

domainId <<*in*>> *(p. 200)* ID of the domain that the application intends to join. **[range]** [>=0], and does not violate guidelines stated in **DDS_RtpsWellKnownPorts_t** (p. 905).

library_name <<*in*>> *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipantFactory::set_default_library** (p. 1225)).

profile_name <<*in*>> *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipantFactory::set_default_profile** (p. 1225)).

listener <<*in*>> *(p. 200)* the DomainParticipant's listener.

mask <<*in*>> *(p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

domain participant or NULL on failure

See also:

Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DomainParticipantQos (p. 588) for rules on consistency among QoS
DDS_PARTICIPANT_QOS_DEFAULT (p. 35)
NDDS_DISCOVERY_PEERS (p. 388)
DDSDomainParticipantFactory::create_participant() (p. 1233)
DDSDomainParticipantFactory::get_default_participant_qos() (p. 1224)
DDSDomainParticipant::set_listener() (p. 1202)

6.185.2.24 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::delete_participant (DDSDomainParticipant * *a_participant*) [pure virtual]

Deletes an existing **DDSDomainParticipant** (p. 1139).

Precondition:

All domain entities belonging to the participant must have already been deleted. Otherwise it fails with the error **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSDomainParticipant** (p. 1139) will not be called after this method returns successfully.

Parameters:

a_participant *<<in>>* *(p. 200)* **DDSDomainParticipant** (p. 1139) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.185.2.25 virtual DDSDomainParticipant* DDSDomainParticipantFactory::lookup_participant (DDS_DomainId_t *domainId*) [pure virtual]

Locates an existing **DDSDomainParticipant** (p. 1139).

If no such **DDSDomainParticipant** (p. 1139) exists, the operation will return NULL value.

If multiple **DDSDomainParticipant** (p. 1139) entities belonging to that domainId exist, then the operation will return one of them. It is not specified which one.

Parameters:

domainId *<<in>> (p. 200)* ID of the domain participant to lookup.

Returns:

domain participant if it exists, or NULL

6.185.2.26 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::set_qos (const DDS_DomainParticipantFactoryQos & *qos*) [pure virtual]

Sets the value for a participant factory QoS.

The **DDS_DomainParticipantFactoryQos::entity_factory** (p. 586) can be changed. The other policies are immutable.

Note that despite having QoS, the **DDSDomainParticipantFactory** (p. 1216) is not an **DDSEntity** (p. 1253).

Parameters:

qos *<<in>> (p. 200)* Set of policies to be applied to **DDSDomainParticipantFactory** (p. 1216). Policies must be consistent. Immutable policies can only be changed before calling any other RTI Connext methods except for **DDSDomainParticipantFactory::get_qos** (p. 1238)

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) if immutable policy is changed, or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) if policies are inconsistent

See also:

DDS_DomainParticipantFactoryQos (p. 586) for rules on consistency among QoS

6.185.2.27 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::get_qos (DDS_DomainParticipantFactoryQos & *qos*) [pure virtual]

Gets the value for participant factory QoS.

Parameters:

qos <<***inout***>> *(p. 200)* QoS to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.28 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::load_profiles () [pure virtual]

<<***eXtension***>> *(p. 199)* Loads the XML QoS profiles.

The XML QoS profiles are loaded implicitly after the first **DDSDomainParticipant** (p. 1139) is created or explicitly, after a call to this method.

This has the same effect as **DDSDomainParticipantFactory::reload_profiles()** (p. 1238).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_ProfileQosPolicy (p. 830)

6.185.2.29 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::reload_profiles () [pure virtual]

<<***eXtension***>> *(p. 199)* Reloads the XML QoS profiles.

The XML QoS profiles are loaded implicitly after the first **DDSDomainParticipant** (p. 1139) is created or explicitly, after a call to this method.

This has the same effect as **DDSDomainParticipantFactory::load_profiles()** (p. 1238).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_ProfileQosPolicy (p. 830)

6.185.2.30 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::unload_profiles () [pure virtual]

*<<**eXtension**>>* (p. *199*) Unloads the XML QoS profiles.

The resources associated with the XML QoS profiles are freed. Any reference to the profiles after calling this method will fail with an error.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_ProfileQosPolicy (p. 830)

6.185.2.31 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::unregister_thread () [pure virtual]

*<<**eXtension**>>* (p. *199*) Allows the user to release thread specific resources kept by the middleware.

This function should be called by the user right before exiting a thread where DDS API were used. In this way the middleware will be able to free all the resources related to this specific thread. The best approach is to call the function during the thread deletion after all the DDS related API have have been called.

Returns:

One of the **Standard Return Codes** (p. 314)

6.185.2.32 virtual DDSDomainParticipant* DDSDomainParticipantFactory::create_participant_from_config_exp (const char * *configuration_name*, const char * *participant_name*) [pure virtual]

*<<**experimental**>>* (p. *199*) *<<**eXtension**>>* (p. *199*) Creates a **DDSDo-**

mainParticipant (p. 1139) given its configuration name from a description provided in an XML configuration file.

This operation creates a **DDSDomainParticipant** (p. 1139) registering all the necessary data types and creating all the contained entities (**DDSTopic** (p. 1419), **DDSPublisher** (p. 1346), **DDSSubscriber** (p. 1390), **DDSDataWriter** (p. 1113), **DDSDataReader** (p. 1087)) from a description given in an XML configuration file.

The configuration name is the fully qualified name of the XML participant object, consisting of the name of the participant library plus the name of participant configuration.

For example the name "MyParticipantLibrary::PublicationParticipant" can be used to create the domain participant from the description in an XML file with contents shown in the snippet below:

```
<participant_library name="MyParticipantLibrary">
     <domain_participant name="PublicationParticipant" domain_ref="MyDomainLibrary::HelloWorldDomain">
         <publisher name="MyPublisher">
             <data_writer name="HelloWorldWriter" topic_ref="HelloWorldTopic"/>
         </publisher>
     </domain_participant>
 </participant_library>
```

The entities belonging to the newly created **DDSDomainParticipant** (p. 1139) can be retrieved with the help pf lookup operations such as: **DDSDomainParticipant::lookup_datareader_by_name_exp** (p. 1214)

Parameters:

configuration_name *<<in>>* *(p. 200)* Name of the participant configuration in the XML file.

participant_name *<<in>>* *(p. 200)* Entity name that is given to the **DDSDomainParticipant** (p. 1139)

Returns:

The created **DDSDomainParticipant** (p. 1139) or NULL on error

See also:

DDSDomainParticipant::lookup_topicdescription (p. 1183)
DDSDomainParticipant::lookup_publisher_by_name_exp (p. 1211)
DDSDomainParticipant::lookup_subscriber_by_name_exp (p. 1212)
DDSDomainParticipant::lookup_datareader_by_name_exp (p. 1214)
DDSDomainParticipant::lookup_datawriter_by_name_exp (p. 1213)

DDSPublisher::lookup_datawriter_by_name_exp (p. 1368)
DDSSubscriber::lookup_datareader_by_name_exp (p. 1412)

6.185.2.33 virtual DDSDomainParticipant* DDSDomainParticipantFactory::lookup_participant_by_name_exp (const char * *participant_name*) [pure virtual]

<<experimental>> (p. 199) *<<eXtension>>* (p. 199) Looks up a **DDSDomainParticipant** (p. 1139) by its entity name in the **DDSDomainParticipantFactory** (p. 1216).

Every **DDSDomainParticipant** (p. 1139) in the system has an entity name which is configured and stored in the EntityName policy, **ENTITY_NAME** (p. 445).

This operation retrieves a **DDSDomainParticipant** (p. 1139) within the **DDSDomainParticipantFactory** (p. 1216) given the entity's name. If there are several **DDSDomainParticipant** (p. 1139) with the same name within the **DDSDomainParticipantFactory** (p. 1216) this function returns the first matching occurrence.

Parameters:

participant_name *<<in>>* (p. 200) Entity name of the **DDSDomainParticipant** (p. 1139).

Returns:

The first **DDSDomainParticipant** (p. 1139) found with the specified name or NULL if it is not found.

6.185.2.34 virtual DDS_ReturnCode_t DDSDomainParticipantFactory::register_type_support_exp (DDSDomainParticipantFactory_RegisterTypeFunction *register_type_fcn*, const char * *type_name*) [pure virtual]

<<experimental>> (p. 199) *<<eXtension>>* (p. 199) Registers a **DDSTypeSupport** (p. 1432) with the **DDSDomainParticipantFactory** (p. 1216) to enable automatic registration if the corresponding type, should it be needed by a **DDSDomainParticipant** (p. 1139).

Types refered by the **DDSTopic** (p. 1419) entities within a **DDSDomainParticipant** (p. 1139) must be registered with the **DDSDomainParticipant** (p. 1139).

Type registration in a **DDSDomainParticipant** (p. 1139) is performed by a call to the **FooTypeSupport::register_type** (p. 1510) operation. This can be done directly from the application code or indirectly by the RTI Connext infrastructure as a result of parsing an XML configuration file that refers to that type.

The **DDSDomainParticipantFactory::register_type_support_exp** (p. 1241) operation provides the **DDSDomainParticipantFactory** (p. 1216) with the information it needs to automatically call the **FooTypeSupport::register_type** (p. 1510) operation and register the corresponding type if the type is needed as a result of parsing an XML configuration file.

Automatic type registration while parsing XML files can also be done by the RTI Connext infrastructure based on the type description provided in the XML files. If the **DDSTypeSupport** (p. 1432) has been registered with the **DDSDomainParticipantFactory** (p. 1216) this definition takes precedence over the description of the type given in the XML file.

The **DDSDomainParticipantFactory::register_type_support_exp** (p. 1241) operation receives a **FooTypeSupport::register_type** (p. 1510) function as a parameter. This function is normally generated using rtiddsgen from a description of the corresponsing type in IDL, XML, or XSD.

The typical workflow when using this function is as follows: Define the datatype in IDL (or XML, or XSD) in a file. E.g. Foo.idl Run rtiddsgen in that file to generate the TypeSuppory files, for the desired programming language. E.g. in C++ FooTypeSupport.h FooTypeSupport.cxx Include the proper header file (e.g. FooTypeSupport.h) in your program and call **DDSDomainParticipantFactory::register_type_support_exp** (p. 1241) passing the function that was generated by rtiddsgen. E.g. **FooTypeSupport::register_type** (p. 1510) Include the TypeSupport source file in your project such that it is compiled and linked. E.g. FooTypeSupport.cxx

You may refer to the `Getting Started Guide` for additional details in this process.

Note that only one register function is allowed per registered type name.

Parameters:

register_type_fcn *<<in>> (p. 200)* DDS_RegisterTypeFunction to be used for registering the type with a **DDSDomainParticipant** (p. 1139).

type_name *<<in>> (p. 200)* Name the type is registered with.

Returns:

One of the **Standard Return Codes** (p. 314)

6.186 DDSDomainParticipantListener Class Reference

<<*interface*>> *(p. 199)* Listener for participant status.

Inheritance diagram for DDSDomainParticipantListener::

6.186.1 Detailed Description

<<*interface*>> *(p. 199)* Listener for participant status.

Entity:

DDSDomainParticipant (p. 1139)

Status:

Status Kinds (p. 317)

This is the interface that can be implemented by an application-provided class and then registered with the **DDSDomainParticipant** (p. 1139) such that the application can be notified by RTI Connext of relevant status changes.

The **DDSDomainParticipantListener** (p. 1243) interface extends all other Listener interfaces and has no additional operation beyond the ones defined by the more general listeners.

The purpose of the **DDSDomainParticipantListener** (p. 1243) is to be the listener of last resort that is notified of all status changes not captured by more specific listeners attached to the **DDSDomainEntity** (p. 1138) objects. When a relevant status change occurs, RTI Connext will first attempt to notify the listener attached to the concerned **DDSDomainEntity** (p. 1138) if one is installed. Otherwise, RTI Connext will notify the Listener attached to the **DDSDomainParticipant** (p. 1139).

Important: Because a **DDSDomainParticipantListener** (p. 1243) may receive callbacks pertaining to many different entities, it is possible for the same listener to receive multiple callbacks simultaneously in different threads. (Such is not the case for listeners of other types.) It is therefore critical that users of this listener provide their own protection for any thread-unsafe activities undertaken in a **DDSDomainParticipantListener** (p. 1243) callback.

Note: Due to a thread-safety issue, the destruction of a DomainParticipantListener from an enabled DomainParticipant should be avoided – even if the DomainParticipantListener has been removed from the DomainParticipant. (This limitation does not affect the Java API.)

See also:

DDSListener (p. 1318)
DDSDomainParticipant::set_listener (p. 1202)

6.187 DDSDynamicDataReader Class Reference

Reads (subscribes to) objects of type **DDS_DynamicData** (p. 622).

Inheritance diagram for DDSDynamicDataReader::

6.187.1 Detailed Description

Reads (subscribes to) objects of type **DDS_DynamicData** (p. 622).

Instantiates `DDSDataReader` (p. 1087) < **DDS_DynamicData** (p. 622) > .

See also:

DDSDataReader (p. 1087)
FooDataReader (p. 1444)
DDS_DynamicData (p. 622)

6.188 DDSDynamicDataTypeSupport Class Reference

A factory for registering a dynamically defined type and creating **DDS_DynamicData** (p. 622) objects.

Inheritance diagram for DDSDynamicDataTypeSupport::

Public Member Functions

- **DDS_Boolean is_valid** ()

  Indicates whether the object was constructed properly.

- **DDS_ReturnCode_t register_type** (**DDSDomainParticipant** *participant, const char *type_name)

  *Associate the **DDS_TypeCode** (p. 992) with the given **DDSDomainParticipant** (p. 1139) under the given logical name.*

- **DDS_ReturnCode_t unregister_type** (**DDSDomainParticipant** *participant, const char *type_name)

  *Remove the definition of this type from the **DDSDomainParticipant** (p. 1139).*

- const char * **get_type_name** () const

  Get the default name of this type.

- const **DDS_TypeCode** * **get_data_type** () const

  *Get the **DDS_TypeCode** (p. 992) wrapped by this **DDSDynamicDataTypeSupport** (p. 1246).*

- **DDS_DynamicData** * **create_data** ()

  *Create a new **DDS_DynamicData** (p. 622) sample initialized with the **DDS_TypeCode** (p. 992) and properties of this **DDSDynamicDataTypeSupport** (p. 1246).*

- **DDS_ReturnCode_t delete_data** (**DDS_DynamicData** *a_data)

  *Finalize and deallocate the **DDS_DynamicData** (p. 622) sample.*

- void **print_data** (const **DDS_DynamicData** *a_data) const

  Print a string representation of the given sample to the given file.

- **DDS_ReturnCode_t copy_data** (**DDS_DynamicData** *dest, const **DDS_DynamicData** *source) const

  Deeply copy the given data samples.

- **DDSDynamicDataTypeSupport** (**DDS_TypeCode** *type, const struct **DDS_DynamicDataTypeProperty_t** &props)

  *Construct a new **DDSDynamicDataTypeSupport*** *(p. 1246)* *object.*

- virtual ~**DDSDynamicDataTypeSupport** ()

  *Delete a **DDSDynamicDataTypeSupport*** *(p. 1246)* *object.*

6.188.1 Detailed Description

A factory for registering a dynamically defined type and creating **DDS_DynamicData** (p. 622) objects.

A **DDSDynamicDataTypeSupport** (p. 1246) has three roles:

1. It associates a **DDS_TypeCode** (p. 992) with policies for managing objects of that type. See the constructor, **DDSDynamicDataTypeSupport::DDSDynamicDataTypeSupport** (p. 1247).
2. It registers its type under logical names with a **DDSDomainParticipant** (p. 1139). See **DDSDynamicDataTypeSupport::register_type** (p. 1249).
3. It creates **DDS_DynamicData** (p. 622) samples pre-initialized with the type and properties of the type support itself. See **DDSDynamicDataTypeSupport::create_data** (p. 1250).

6.188.2 Constructor & Destructor Documentation

6.188.2.1 DDSDynamicDataTypeSupport::DDSDynamicDataTypeSupport (DDS_TypeCode * *type*, const struct DDS_DynamicDataTypeProperty_t & *props*)

Construct a new **DDSDynamicDataTypeSupport** (p. 1246) object.

This step is usually followed by type registration.

NOTE that RTI Connext does not explicitly generate any exceptions in this constructor, because C++ exception support is not consistent across all platforms on which RTI Connext runs. Therefore, to check whether construction succeeded, you must use the **DDSDynamicDataTypeSupport::is_valid** (p. 1248) method.

The **DDS_TypeCode** (p. 992) object that is passed to this constructor is cloned and stored internally; no pointer is retained to the object passed in. It is therefore safe to delete the **DDS_TypeCode** (p. 992) after this method returns.

Parameters:

type The **DDS_TypeCode** (p. 992) that describes the members of this type. The new object will contain a *copy* of this **DDS_TypeCode** (p. 992); you may delete the argument after this constructor returns.

props Policies that describe how to manage the memory and other properties of the data samples created by this factory. In most cases, the default values will be appropriate; see **DDS_DYNAMIC_DATA_TYPE_PROPERTY_DEFAULT** (p. 81).

See also:

DDSDynamicDataTypeSupport::is_valid (p. 1248)
DDSDynamicDataTypeSupport::register_type (p. 1249)
DDSDynamicDataTypeSupport::~DDSDynamicDataTypeSupport (p. 1248)

6.188.2.2 virtual DDSDynamicDataTypeSupport::~DDSDynamicDataTypeSupport () [virtual]

Delete a **DDSDynamicDataTypeSupport** (p. 1246) object.

A **DDSDynamicDataTypeSupport** (p. 1246) cannot be deleted while it is still in use. For each **DDSDomainParticipant** (p. 1139) with which the **DDSDynamicDataTypeSupport** (p. 1246) is registered, either the type must be unregistered or the participant must be deleted.

See also:

DDSDynamicDataTypeSupport::unregister_type (p. 1249)
DDSDynamicDataTypeSupport::DDSDynamicDataTypeSupport (p. 1247)

6.188.3 Member Function Documentation

6.188.3.1 DDS_Boolean DDSDynamicDataTypeSupport::is_valid ()

Indicates whether the object was constructed properly.

This method returns **DDS_BOOLEAN_TRUE** (p. 298) if the constructor succeeded; it returns **DDS_BOOLEAN_FALSE** (p. 299) if the constructor failed for any reason, which should also have resulted in a log message.

Possible failure reasons include passing an invalid type or invalid properties to the constructor.

This method is necessary because C++ exception support is not consistent across all of the platforms on which RTI Connext runs. Therefore, the implementation does not throw any exceptions in the constructor.

See also:

DDSDynamicDataTypeSupport::DDSDynamicDataTypeSupport (p. 1247)

6.188.3.2 DDS_ReturnCode_t DDSDynamicDataTypeSupport::register_type (DDSDomainParticipant * *participant*, const char * *type_name*)

Associate the **DDS_TypeCode** (p. 992) with the given **DDSDomainParticipant** (p. 1139) under the given logical name.

Once a type has been registered, it can be referenced by name when creating a topic. Statically and dynamically defined types behave the same way in this respect.

See also:

FooTypeSupport::register_type (p. 1510)
DDSDomainParticipant::create_topic (p. 1175)
DDSDynamicDataTypeSupport::unregister_type (p. 1249)

6.188.3.3 DDS_ReturnCode_t DDSDynamicDataTypeSupport::unregister_type (DDSDomainParticipant * *participant*, const char * *type_name*)

Remove the definition of this type from the **DDSDomainParticipant** (p. 1139).

This operation is optional; all types are automatically unregistered when a **DDSDomainParticipant** (p. 1139) is deleted. Most application will not need to manually unregister types.

A type cannot be unregistered while it is still in use; that is, while any **DDSTopic** (p. 1419) is still referring to it.

See also:

FooTypeSupport::unregister_type (p. 1511)
DDSDynamicDataTypeSupport::register_type (p. 1249)

6.188.3.4 const char* DDSDynamicDataTypeSupport::get_type_name () const

Get the default name of this type.

The **DDS_TypeCode** (p. 992) that is wrapped by this **DDSDynamicDataTypeSupport** (p. 1246) includes a name; this operation returns that name.

This operation is useful when registering a type, because in most cases it is not necessary for the physical and logical names of the type to be different.

```
myTypeSupport->register_type(myParticipant, myTypeSupport->get_type_name());
```

See also:

FooTypeSupport::get_type_name (p. 1517)

6.188.3.5 const DDS_TypeCode* DDSDynamicDataTypeSupport::get_data_type () const

Get the **DDS_TypeCode** (p. 992) wrapped by this **DDSDynamicDataTypeSupport** (p. 1246).

6.188.3.6 DDS_DynamicData* DDSDynamicDataTypeSupport::create_data ()

Create a new **DDS_DynamicData** (p. 622) sample initialized with the **DDS_TypeCode** (p. 992) and properties of this **DDSDynamicDataTypeSupport** (p. 1246).

You must delete your **DDS_DynamicData** (p. 622) object when you are finished with it.

```
DDS_DynamicData* sample = myTypeSupport->create_data();
 // Do something...
 myTypeSupport->delete_data(sample);
```

See also:

DDSDynamicDataTypeSupport::delete_data (p. 1251)
FooTypeSupport::create_data (p. 1512)
DDS_DynamicData::DDS_DynamicData
DDS_DynamicDataTypeProperty_t::data (p. 728)

6.188.3.7 DDS_ReturnCode_t DDSDynamicDataTypeSupport::delete_data (DDS_DynamicData * *a_data*)

Finalize and deallocate the **DDS_DynamicData** (p. 622) sample.

See also:

FooTypeSupport::delete_data (p. 1514)
DDSDynamicDataTypeSupport::create_data (p. 1250)

6.188.3.8 void DDSDynamicDataTypeSupport::print_data (const DDS_DynamicData * *a_data*) const

Print a string representation of the given sample to the given file.

This method is equivalent to **DDS_DynamicData::print** (p. 645).

See also:

DDS_DynamicData::print (p. 645)

6.188.3.9 DDS_ReturnCode_t DDSDynamicDataTypeSupport::copy_data (DDS_DynamicData * *dest*, const DDS_DynamicData * *source*) const

Deeply copy the given data samples.

6.189 DDSDynamicDataWriter Class Reference

Writes (publishes) objects of type **DDS_DynamicData** (p. 622).

Inheritance diagram for DDSDynamicDataWriter::

6.189.1 Detailed Description

Writes (publishes) objects of type **DDS_DynamicData** (p. 622).

Instantiates `DDSDataWriter` (p. 1113) < **DDS_DynamicData** (p. 622) > .

See also:

DDSDataWriter (p. 1113)
FooDataWriter (p. 1475)
DDS_DynamicData (p. 622)

6.190 DDSEntity Class Reference

<<***interface***>> *(*p. *199)* Abstract base class for all the DDS objects that support QoS policies, a listener, and a status condition.

Inheritance diagram for DDSEntity::

Public Member Functions

- virtual **DDS_ReturnCode_t enable** ()=0

  Enables the ***DDSEntity*** *(*p. *1253).*

- virtual **DDSStatusCondition** * **get_statuscondition** ()=0

  Allows access to the ***DDSStatusCondition*** *(*p. *1376) associated with the* ***DDSEntity*** *(*p. *1253).*

- virtual **DDS_StatusMask get_status_changes** ()=0

  Retrieves the list of communication statuses in the ***DDSEntity*** *(*p. *1253) that are triggered.*

- virtual **DDS_InstanceHandle_t get_instance_handle** ()=0

  Allows access to the ***DDS_InstanceHandle_t*** *(*p. *53) associated with the* ***DDSEntity*** *(*p. *1253).*

6.190.1 Detailed Description

<<***interface***>> *(*p. *199)* Abstract base class for all the DDS objects that support QoS policies, a listener, and a status condition.

All operations except for set_qos(), get_qos(), set_listener(), get_listener() and **enable()** (p. 1256), may return the value **DDS_RETCODE_NOT_ENABLED** (p. 315).

QoS:

QoS Policies (p. 331)

Status:

Status Kinds (p. 317)

Listener:

DDSListener (p. 1318)

6.190.2 Abstract operations

Each derived entity provides the following operations specific to its role in RTI Connext.

6.190.2.1 set_qos (abstract)

This operation sets the QoS policies of the **DDSEntity** (p. 1253).

This operation must be provided by each of the derived **DDSEntity** (p. 1253) classes (**DDSDomainParticipant** (p. 1139), **DDSTopic** (p. 1419), **DDSPublisher** (p. 1346), **DDSDataWriter** (p. 1113), **DDSSubscriber** (p. 1390), and **DDSDataReader** (p. 1087)) so that the policies that are meaningful to each **DDSEntity** (p. 1253) can be set.

Precondition:

Certain policies are immutable (see **QoS Policies** (p. 331)): they can only be set at **DDSEntity** (p. 1253) creation time or before the entity is enabled. If `set_qos()` is invoked after the **DDSEntity** (p. 1253) is enabled and it attempts to change the value of an immutable policy, the operation will fail and return **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315).
Certain values of QoS policies can be incompatible with the settings of the other policies. The `set_qos()` operation will also fail if it specifies a set of values that, once combined with the existing values, would result in an inconsistent set of policies. In this case, the operation will fail and return **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).
If the application supplies a non-default value for a QoS policy that is not supported by the implementation of the service, the set_qos operation will fail and return **DDS_RETCODE_UNSUPPORTED** (p. 315).

Postcondition:

The existing set of policies is only changed if the `set_qos()` operation succeeds. This is indicated by a return code of **DDS_RETCODE_OK** (p. 315). In all other cases, none of the policies are modified.

Each derived **DDSEntity** (p. 1253) class (**DDSDomainParticipant** (p. 1139), **DDSTopic** (p. 1419), **DDSPublisher** (p. 1346), **DDSDataWriter** (p. 1113), **DDSSubscriber** (p. 1390), **DDSDataReader** (p. 1087)) has a corresponding special value of the QoS (**DDS_PARTICIPANT_QOS_DEFAULT** (p. 35), **DDS_PUBLISHER_QOS_DEFAULT** (p. 38), **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39), **DDS_TOPIC_QOS_DEFAULT** (p. 38), **DDS_DATAWRITER_QOS_DEFAULT** (p. 84), **DDS_DATAREADER_QOS_DEFAULT** (p. 99)). This special value may be used as a parameter to the `set_qos` operation to indicate that the QoS of the **DDSEntity** (p. 1253)

should be changed to match the current default QoS set in the **DDSEntity** (p. 1253)'s factory. The operation set_qos cannot modify the immutable QoS, so a successful return of the operation indicates that the mutable QoS for the Entity has been modified to match the current default for the **DDSEntity** (p. 1253)'s factory.

The set of policies specified in the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set.

Possible error codes returned in addition to **Standard Return Codes** (p. 314) : **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

6.190.2.2 get_qos (abstract)

This operation allows access to the existing set of QoS policies for the **DDSEntity** (p. 1253). This operation must be provided by each of the derived **DDSEntity** (p. 1253) classes (**DDSDomainParticipant** (p. 1139), **DDSTopic** (p. 1419), **DDSPublisher** (p. 1346), **DDSDataWriter** (p. 1113), **DDSSubscriber** (p. 1390), and **DDSDataReader** (p. 1087)), so that the policies that are meaningful to each **DDSEntity** (p. 1253) can be retrieved.

Possible error codes are **Standard Return Codes** (p. 314).

6.190.2.3 set_listener (abstract)

This operation installs a **DDSListener** (p. 1318) on the **DDSEntity** (p. 1253). The listener will only be invoked on the changes of communication status indicated by the specified `mask`.

This operation must be provided by each of the derived **DDSEntity** (p. 1253) classes (**DDSDomainParticipant** (p. 1139), **DDSTopic** (p. 1419), **DDSPublisher** (p. 1346), **DDSDataWriter** (p. 1113), **DDSSubscriber** (p. 1390), and **DDSDataReader** (p. 1087)), so that the listener is of the concrete type suitable to the particular **DDSEntity** (p. 1253).

It is permitted to use NULL as the value of the listener. The NULL listener behaves as if the `mask` is **DDS_STATUS_MASK_NONE** (p. 321).

Postcondition:

Only one listener can be attached to each **DDSEntity** (p. 1253). If a listener was already set, the operation `set_listener()` will replace it with the new one. Consequently, if the value NULL is passed for the listener parameter to the set_listener operation, any existing listener will be removed.

6.190.2.4 get_listener (abstract)

This operation allows access to the existing **DDSListener** (p. 1318) attached to the **DDSEntity** (p. 1253).

This operation must be provided by each of the derived **DDSEntity** (p. 1253) classes (**DDSDomainParticipant** (p. 1139), **DDSTopic** (p. 1419), **DDSPublisher** (p. 1346), **DDSDataWriter** (p. 1113), **DDSSubscriber** (p. 1390), and **DDSDataReader** (p. 1087)) so that the listener is of the concrete type suitable to the particular **DDSEntity** (p. 1253).

If no listener is installed on the **DDSEntity** (p. 1253), this operation will return NULL.

6.190.3 Member Function Documentation

6.190.3.1 virtual DDS_ReturnCode_t DDSEntity::enable () [pure virtual]

Enables the **DDSEntity** (p. 1253).

This operation enables the Entity. Entity objects can be created either enabled or disabled. This is controlled by the value of the **ENTITY_FACTORY** (p. 377) QoS policy on the corresponding factory for the **DDSEntity** (p. 1253).

By default, **ENTITY_FACTORY** (p. 377) is set so that it is not necessary to explicitly call **DDSEntity::enable** (p. 1256) on newly created entities.

The **DDSEntity::enable** (p. 1256) operation is idempotent. Calling enable on an already enabled Entity returns OK and has no effect.

If a **DDSEntity** (p. 1253) has not yet been enabled, the following kinds of operations may be invoked on it:

- set or get the QoS policies (including default QoS policies) and listener
- **DDSEntity::get_statuscondition** (p. 1257)
- 'factory' operations
- **DDSEntity::get_status_changes** (p. 1257) and other get status operations (although the status of a disabled entity never changes)
- 'lookup' operations

Other operations may explicitly state that they may be called on disabled entities; those that do not will return the error **DDS_RETCODE_NOT_ENABLED** (p. 315).

It is legal to delete an **DDSEntity** (p. 1253) that has not been enabled by calling the proper operation on its factory.

Entities created from a factory that is disabled are created disabled, regardless of the setting of the **DDS_EntityFactoryQosPolicy** (p. 733).

Calling enable on an Entity whose factory is not enabled will fail and return **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

If **DDS_EntityFactoryQosPolicy::autoenable_created_entities** (p. 734) is TRUE, the enable operation on a factory will automatically enable all entities created from that factory.

Listeners associated with an entity are not called until the entity is enabled.

Conditions associated with a disabled entity are "inactive," that is, they have a `trigger_value` == FALSE.

Returns:

One of the **Standard Return Codes** (p. 314), **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Implemented in **DDSDataWriter** (p. 1130), and **DDSDataReader** (p. 1105).

6.190.3.2 virtual DDSStatusCondition* DDSEntity::get_statuscondition () [pure virtual]

Allows access to the **DDSStatusCondition** (p. 1376) associated with the **DDSEntity** (p. 1253).

The returned condition can then be added to a **DDSWaitSet** (p. 1433) so that the application can wait for specific status changes that affect the **DDSEntity** (p. 1253).

Returns:

the status condition associated with this entity.

Implemented in **DDSDataWriter** (p. 1131), and **DDSDataReader** (p. 1106).

6.190.3.3 virtual DDS_StatusMask DDSEntity::get_status_changes () [pure virtual]

Retrieves the list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.

That is, the list of statuses whose value has changed since the last time the application read the status using the get_*_status() method.

When the entity is first created or if the entity is not enabled, all communication statuses are in the "untriggered" state so the list returned by the get_status_changes operation will be empty.

The list of statuses returned by the get_status_changes operation refers to the status that are triggered on the Entity itself and does not include statuses that apply to contained entities.

Returns:

list of communication statuses in the **DDSEntity** (p. 1253) that are triggered.

See also:

Status Kinds (p. 317)

Implemented in **DDSDataWriter** (p. 1131), and **DDSDataReader** (p. 1106).

6.190.3.4 virtual DDS_InstanceHandle_t DDSEntity::get_instance_handle () [pure virtual]

Allows access to the **DDS_InstanceHandle_t** (p. 53) associated with the **DDSEntity** (p. 1253).

This operation returns the **DDS_InstanceHandle_t** (p. 53) that represents the **DDSEntity** (p. 1253).

Returns:

the instance handle associated with this entity.

Implemented in **DDSDataWriter** (p. 1132), and **DDSDataReader** (p. 1107).

6.191 DDSFlowController Class Reference

<<*interface*>> *(p. 199)* A flow controller is the object responsible for shaping the network traffic by determining when attached asynchronous **DDSDataWriter** (p. 1113) instances are allowed to write data.

Public Member Functions

- virtual **DDS_ReturnCode_t set_property** (const struct **DDS_FlowControllerProperty_t** &prop)=0

  *Sets the **DDSFlowController** (p. 1259) property.*

- virtual **DDS_ReturnCode_t get_property** (struct **DDS_FlowControllerProperty_t** &prop)=0

  *Gets the **DDSFlowController** (p. 1259) property.*

- virtual **DDS_ReturnCode_t trigger_flow** ()=0

  *Provides an external trigger to the **DDSFlowController** (p. 1259).*

- virtual const char * **get_name** ()=0

  *Returns the name of the **DDSFlowController** (p. 1259).*

- virtual **DDSDomainParticipant** * **get_participant** ()=0

  *Returns the **DDSDomainParticipant** (p. 1139) to which the **DDSFlowController** (p. 1259) belongs.*

6.191.1 Detailed Description

<<*interface*>> *(p. 199)* A flow controller is the object responsible for shaping the network traffic by determining when attached asynchronous **DDSDataWriter** (p. 1113) instances are allowed to write data.

QoS:

DDS_FlowControllerProperty_t (p. 749)

6.191.2 Member Function Documentation

6.191.2.1 virtual DDS_ReturnCode_t DDSFlowController::set_property (const struct DDS_FlowControllerProperty_t & *prop*) [pure virtual]

Sets the **DDSFlowController** (p. 1259) property.

This operation modifies the property of the **DDSFlowController** (p. 1259).

Once a **DDSFlowController** (p. 1259) has been instantiated, only the **DDS_FlowControllerProperty_t::token_bucket** (p. 750) can be changed. The **DDS_FlowControllerProperty_t::scheduling_policy** (p. 750) is immutable.

A new **DDS_FlowControllerTokenBucketProperty_t::period** (p. 753) only takes effect at the next scheduled token distribution time (as determined by its previous value).

Parameters:

prop *<<in>> (p. 200)* The new **DDS_FlowControllerProperty_t** (p. 749). Property must be consistent. Immutable fields cannot be changed after **DDSFlowController** (p. 1259) has been created. The special value **DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT** (p. 40) can be used to indicate that the property of the **DDSFlowController** (p. 1259) should be changed to match the current default **DDS_FlowControllerProperty_t** (p. 749) set in the **DDSDomainParticipant** (p. 1139).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_FlowControllerProperty_t (p. 749) for rules on consistency among property values.

6.191.2.2 virtual DDS_ReturnCode_t DDSFlowController::get_property (struct DDS_FlowControllerProperty_t & *prop*) [pure virtual]

Gets the **DDSFlowController** (p. 1259) property.

Parameters:

prop *<<**in**>>* *(p. 200)* **DDSFlowController** (p. 1259) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

6.191.2.3 virtual DDS_ReturnCode_t DDSFlowController::trigger_flow () [pure virtual]

Provides an external trigger to the **DDSFlowController** (p. 1259).

Typically, a **DDSFlowController** (p. 1259) uses an internal trigger to periodically replenish its tokens. The period by which this trigger is called is determined by the **DDS_FlowControllerTokenBucketProperty_t::period** (p. 753) property setting.

This function provides an additional, external trigger to the **DDSFlowController** (p. 1259). This trigger adds **DDS_FlowControllerTokenBucketProperty_t::tokens_added_per_period** (p. 752) tokens each time it is called (subject to the other property settings of the **DDSFlowController** (p. 1259)).

An *on-demand* **DDSFlowController** (p. 1259) can be created with a **DDS_DURATION_INFINITE** (p. 305) as **DDS_FlowControllerTokenBucketProperty_t::period** (p. 753), in which case the only trigger source is external (i.e. the **DDSFlowController** (p. 1259) is solely triggered by the user on demand).

DDSFlowController::trigger_flow (p. 1261) can be called on both strict *on-demand* **DDSFlowController** (p. 1259) and hybrid **DDSFlowController** (p. 1259) (internally and externally triggered).

Returns:

One of the **Standard Return Codes** (p. 314)

6.191.2.4 virtual const char* DDSFlowController::get_name () [pure virtual]

Returns the name of the **DDSFlowController** (p. 1259).

Returns:

The name of the **DDSFlowController** (p. 1259).

6.191.2.5 virtual DDSDomainParticipant* DDSFlowController::get_participant () [pure virtual]

Returns the **DDSDomainParticipant** (p. 1139) to which the **DDSFlowController** (p. 1259) belongs.

Returns:

The **DDSDomainParticipant** (p. 1139) to which the **DDSFlowController** (p. 1259) belongs.

6.192 DDSGuardCondition Class Reference

<<*interface*>> *(p. 199)* A specific **DDSCondition** (p. 1075) whose `trigger_value` is completely under the control of the application.

Inheritance diagram for DDSGuardCondition::

Public Member Functions

- virtual **DDS_Boolean get_trigger_value** ()
- virtual **DDS_ReturnCode_t set_trigger_value** (**DDS_Boolean** value)

  Set the guard condition trigger value.

- **DDSGuardCondition** ()

  No argument constructor.

- virtual ~**DDSGuardCondition** ()

  Destructor.

6.192.1 Detailed Description

<<*interface*>> *(p. 199)* A specific **DDSCondition** (p. 1075) whose `trigger_value` is completely under the control of the application.

The **DDSGuardCondition** (p. 1263) provides a way for an application to manually wake up a **DDSWaitSet** (p. 1433). This is accomplished by attaching the **DDSGuardCondition** (p. 1263) to the **DDSWaitSet** (p. 1433) and then setting the `trigger_value` by means of the **DDSGuardCondition::set_trigger_value** (p. 1264) operation.

See also:

DDSWaitSet (p. 1433)

6.192.2 Constructor & Destructor Documentation

6.192.2.1 DDSGuardCondition::DDSGuardCondition ()

No argument constructor.

Construct a new guard condition on the heap.

Returns:

A new condition with trigger value **DDS_BOOLEAN_FALSE** (p. 299), or NULL if a condition could not be allocated.

6.192.2.2 virtual DDSGuardCondition::~DDSGuardCondition () [virtual]

Destructor.

Releases the resources associated with this object.

Deleting a NULL condition is safe and has no effect.

6.192.3 Member Function Documentation

6.192.3.1 virtual DDS_Boolean DDSGuardCondition::get_trigger_value () [virtual]

Retrieve the `trigger_value`.

Returns:

the trigger value.

Implements **DDSCondition** (p. 1075).

6.192.3.2 virtual DDS_ReturnCode_t DDSGuardCondition::set_trigger_value (DDS_Boolean *value*) [virtual]

Set the guard condition trigger value.

Parameters:

value *<<in>>* *(p. 200)* the new trigger value.

6.193 DDSKeyedOctetsDataReader Class Reference

<<*interface*>> *(p. 199)* Instantiates `DataReader` < **DDS_KeyedOctets** (p. 765) >.

Inheritance diagram for DDSKeyedOctetsDataReader::

Public Member Functions

- virtual **DDS_ReturnCode_t** **read** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t** **take** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data-samples from the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t** **read_w_condition** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  *Accesses via **DDSKeyedOctetsDataReader::read** (p. 1269) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).*

- virtual **DDS_ReturnCode_t** **take_w_condition** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  *Analogous to **DDSKeyedOctetsDataReader::read_w_condition** (p. 1270) except it accesses samples via the **DDSKeyedOctetsDataReader::take** (p. 1269) operation.*

ˆ virtual **DDS_ReturnCode_t read_next_sample** (**DDS_KeyedOctets** &received_data, **DDS_SampleInfo** &sample_info)

*Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).*

ˆ virtual **DDS_ReturnCode_t take_next_sample** (**DDS_KeyedOctets** &received_data, **DDS_SampleInfo** &sample_info)

*Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).*

ˆ virtual **DDS_ReturnCode_t read_instance** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

*Access a collection of data samples from the **DDSDataReader** (p. 1087).*

ˆ virtual **DDS_ReturnCode_t take_instance** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

*Access a collection of data samples from the **DDSDataReader** (p. 1087).*

ˆ virtual **DDS_ReturnCode_t read_instance_w_condition** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

*Accesses via **DDSKeyedOctetsDataReader::read_instance** (p. 1271) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).*

ˆ virtual **DDS_ReturnCode_t take_instance_w_condition** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

*Accesses via **DDSKeyedOctetsDataReader::take_instance*** (p. 1271) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t** **read_next_instance** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader*** (p. 1087).

- virtual **DDS_ReturnCode_t** **take_next_instance** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader*** (p. 1087).

- virtual **DDS_ReturnCode_t** **read_next_instance_w_condition** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

  *Accesses via **DDSKeyedOctetsDataReader::read_next_instance*** (p. 1272) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t** **take_next_instance_w_condition** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

  *Accesses via **DDSKeyedOctetsDataReader::take_next_instance*** (p. 1273) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t** **return_loan** (**DDS_KeyedOctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq)

*Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of* `received_data` *and* `info_seq` *obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t get_key_value** (**DDS_KeyedOctets** &key_holder, const **DDS_InstanceHandle_t** &handle)

  Retrieve the instance `key` *that corresponds to an instance* `handle`*.*

- virtual **DDS_ReturnCode_t get_key_value** (char *key, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `key` *that corresponds to an instance* `handle`*.*

- virtual **DDS_InstanceHandle_t lookup_instance** (const **DDS_KeyedOctets** &key_holder)

  Retrieve the instance `handle` *that corresponds to an instance* `key_holder`*.*

- virtual **DDS_InstanceHandle_t lookup_instance** (const char *key)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `handle` *that corresponds to an instance* `key`*.*

Static Public Member Functions

- static **DDSKeyedOctetsDataReader** * **narrow** (**DDSDataReader** *reader)

  *Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSKeyedOctetsDataReader** (p. 1265) pointer.*

6.193.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataReader` < **DDS_KeyedOctets** (p. 765) >.

When reading or taking data with this reader, if you request a copy of the samples instead of a loan, and a string or byte array in a destination data sample is NULL, the middleware will allocate a new string or array for you of sufficient length to hold the received data. New strings will be allocated with **DDS_String_alloc** (p. 458); new arrays will be allocated with **DDS_OctetBuffer_alloc** (p. 453). The sample's destructor will delete them.

A non- NULL string or array is assumed to be allocated to sufficient length to store the incoming data. It will not be reallocated.

See also:

FooDataReader (p. 1444)
DDSDataReader (p. 1087)

6.193.2 Member Function Documentation

6.193.2.1 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read (p. 1447)

6.193.2.2 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data-samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take (p. 1448)

6.193.2.3 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read_w_condition (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Accesses via **DDSKeyedOctetsDataReader::read** (p. 1269) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_w_condition (p. 1454)

6.193.2.4 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take_w_condition (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Analogous to **DDSKeyedOctetsDataReader::read_w_condition** (p. 1270) except it accesses samples via the **DDSKeyedOctetsDataReader::take** (p. 1269) operation.

See also:

FooDataReader::take_w_condition (p. 1456)

6.193.2.5 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read_next_sample (DDS_KeyedOctets & *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_next_sample (p. 1457)

6.193.2.6 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take_next_sample (DDS_KeyedOctets & *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_next_sample (p. 1458)

6.193.2.7 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read_instance (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_instance (p. 1459)

6.193.2.8 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take_instance (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_instance (p. 1460)

6.193.2.9 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read_instance_w_condition (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedOctetsDataReader::read_instance** (p. 1271) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_instance_w_condition (p. 1462)

6.193.2.10 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take_instance_w_condition (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedOctetsDataReader::take_instance** (p. 1271) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::take_instance_w_condition (p. 1463)

6.193.2.11 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read_next_instance (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_next_instance (p. 1464)

6.193.2.12 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take_next_instance (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_next_instance (p. 1467)

6.193.2.13 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::read_next_instance_w_condition (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedOctetsDataReader::read_next_instance** (p. 1272) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_next_instance_w_condition (p. 1468)

6.193.2.14 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::take_next_instance_w_condition (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedOctetsDataReader::take_next_instance** (p. 1273) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::take_next_instance_w_condition (p. 1470)

6.193.2.15 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::return_loan (DDS_KeyedOctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*) [virtual]

Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of `received_data` and `info_seq` obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).

See also:

FooDataReader::return_loan (p. 1471)

6.193.2.16 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::get_key_value (DDS_KeyedOctets & *key_holder*, const DDS_InstanceHandle_t & *handle*) [virtual]

Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataReader::get_key_value (p. 1472)

6.193.2.17 virtual DDS_ReturnCode_t DDSKeyedOctetsDataReader::get_key_value (char * *key*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<*eXtension*>> *(p. 199)* Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataReader::get_key_value (p. 1472)

6.193.2.18 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataReader::lookup_instance (const DDS_KeyedOctets & *key_holder*) [virtual]

Retrieve the instance `handle` that corresponds to an instance `key_holder`.

See also:

FooDataReader::lookup_instance (p. 1473)

6.193.2.19 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataReader::lookup_instance (const char * *key*) [virtual]

<<*eXtension*>> *(p. 199)* Retrieve the instance `handle` that corresponds to an instance `key`.

See also:

FooDataReader::lookup_instance (p. 1473)

6.193.2.20 static DDSKeyedOctetsDataReader* DDSKeyedOctetsDataReader::narrow (DDSDataReader * *reader*) [static]

Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSKeyedOctetsDataReader** (p. 1265) pointer.

See also:

FooDataReader::narrow (p. 1447)

6.194 DDSKeyedOctetsDataWriter Class Reference

<<*interface*>> *(p. 199)* Instantiates `DataWriter` < **DDS_KeyedOctets** (p. 765) >.

Inheritance diagram for DDSKeyedOctetsDataWriter::

Public Member Functions

- virtual **DDS_InstanceHandle_t register_instance** (const **DDS_KeyedOctets** &instance_data)

  Informs RTI Connext that the application will be modifying a particular instance.

- virtual **DDS_InstanceHandle_t register_instance** (const char *key)

  <<**eXtension**>> (p. 199) *Informs RTI Connext that the application will be modifying a particular instance.*

- virtual **DDS_InstanceHandle_t register_instance_w_timestamp** (const **DDS_KeyedOctets** &instance_data, const **DDS_Time_t** &source_timestamp)

  *Performs the same functions as **DDSKeyedOctetsDataWriter::register_instance** (p. 1280) except that the application provides the value for the* `source_timestamp`.

- virtual **DDS_InstanceHandle_t register_instance_w_timestamp** (const char *key, const **DDS_Time_t** &source_timestamp)

  <<**eXtension**>> (p. 199) *Performs the same functions as **DDSKeyedOctetsDataWriter::register_instance** (p. 1280) except that the application provides the value for the* `source_timestamp`.

- virtual **DDS_ReturnCode_t unregister_instance** (const **DDS_KeyedOctets** &instance_data, const **DDS_InstanceHandle_t** &handle)

  *Reverses the action of **DDSKeyedOctetsDataWriter::register_instance** (p. 1280).*

- virtual **DDS_ReturnCode_t unregister_instance** (const char *key, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Reverses the action of **DDSKeyedOctetsDataWriter::register_instance** (p. 1280).*

ˆ virtual **DDS_ReturnCode_t unregister_instance_w_timestamp** (const **DDS_KeyedOctets** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

*Performs the same function as **DDSKeyedOctetsDataWriter::unregister_instance*** (p. 1281) *except that it also provides the value for the* `source_timestamp`.

ˆ virtual **DDS_ReturnCode_t unregister_instance_w_timestamp** (const char *key, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

*Performs the same function as **DDSKeyedOctetsDataWriter::unregister_instance*** (p. 1281) *except that it also provides the value for the* `source_timestamp`.

ˆ virtual **DDS_ReturnCode_t write** (const **DDS_KeyedOctets** &instance_data, const **DDS_InstanceHandle_t** &handle)

*Modifies the value of a **DDS_KeyedOctets*** (p. 765) *data instance.*

ˆ virtual **DDS_ReturnCode_t write** (const char *key, const unsigned char *octets, int length, const **DDS_InstanceHandle_t** &handle)

<<**eXtension**>> (p. 199) *Modifies the value of a **DDS_KeyedOctets*** (p. 765) *data instance.*

ˆ virtual **DDS_ReturnCode_t write** (const char *key, const **DDS_OctetSeq** &octets, const **DDS_InstanceHandle_t** &handle)

<<**eXtension**>> (p. 199) *Modifies the value of a **DDS_KeyedOctets*** (p. 765) *data instance.*

ˆ virtual **DDS_ReturnCode_t write_w_timestamp** (const **DDS_KeyedOctets** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

*Performs the same function as **DDSKeyedOctetsDataWriter::write*** (p. 1282) *except that it also provides the value for the* `source_timestamp`.

ˆ virtual **DDS_ReturnCode_t write_w_timestamp** (const char *key, const unsigned char *octets, int length, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

*Performs the same function as **DDSKeyedOctetsDataWriter::write*** (p. 1282) *except that it also provides the value for the* `source_timestamp`.

ˆ virtual **DDS_ReturnCode_t write_w_timestamp** (const char *key, const **DDS_OctetSeq** &octets, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

Performs the same function as ***DDSKeyedOctetsDataWriter::write*** *(*p. 1282*) except that it also provides the value for the* `source_timestamp`*.*

ˆ virtual **DDS_ReturnCode_t write_w_params** (const **DDS_KeyedOctets** &instance_data, **DDS_WriteParams_t** &params)

Performs the same function as ***DDSKeyedOctetsDataWriter::write*** *(*p. 1282*) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

ˆ virtual **DDS_ReturnCode_t write_w_params** (const char *key, const unsigned char *octets, int length, **DDS_WriteParams_t** &params)

Performs the same function as ***DDSKeyedOctetsDataWriter::write*** *(*p. 1282*) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

ˆ virtual **DDS_ReturnCode_t write_w_params** (const char *key, const **DDS_OctetSeq** &octets, **DDS_WriteParams_t** &params)

Performs the same function as ***DDSKeyedOctetsDataWriter::write*** *(*p. 1282*) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

ˆ virtual **DDS_ReturnCode_t dispose** (const **DDS_KeyedOctets** &instance_data, const **DDS_InstanceHandle_t** &handle)

Requests the middleware to delete the data.

ˆ virtual **DDS_ReturnCode_t dispose** (const char *key, const **DDS_InstanceHandle_t** &instance_handle)

<<**eXtension**>> (p. 199) *Requests the middleware to delete the data.*

ˆ virtual **DDS_ReturnCode_t dispose_w_timestamp** (const **DDS_KeyedOctets** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

Performs the same functions as ***DDSKeyedOctetsDataWriter::dispose*** *(*p. 1286*) except that the application provides the value for the* `source_timestamp` *that is made available to* ***DDSDataReader*** *(*p. 1087*) objects by means of the* `source_timestamp` *attribute inside the* ***DDS_SampleInfo*** *(*p. 912*).*

ˆ virtual **DDS_ReturnCode_t dispose_w_timestamp** (const char *key, const **DDS_InstanceHandle_t** &instance_handle, const **DDS_Time_t** &source_timestamp)

<<**eXtension**>> (p. 199) *Performs the same functions as* ***DDSKeyedOctetsDataWriter::dispose*** *(*p. 1286*) except that the application provides the value for the* `source_timestamp` *that is made available to* ***DDSDataReader*** *(*p. 1087*) objects by means of the* `source_timestamp` *attribute inside the* ***DDS_SampleInfo*** *(*p. 912*).*

- virtual **DDS_ReturnCode_t get_key_value** (**DDS_KeyedOctets** &key_holder, const **DDS_InstanceHandle_t** &handle)

  Retrieve the instance `key` *that corresponds to an instance* `handle`.

- virtual **DDS_ReturnCode_t get_key_value** (char *key, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `key` *that corresponds to an instance* `handle`.

- virtual **DDS_InstanceHandle_t lookup_instance** (const **DDS_KeyedOctets** &key_holder)

  Retrieve the instance `handle` *that corresponds to an instance* `key_holder`.

- virtual **DDS_InstanceHandle_t lookup_instance** (const char *key)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `handle` *that corresponds to an instance* `key`.

Static Public Member Functions

- static **DDSKeyedOctetsDataWriter** * **narrow** (**DDSDataWriter** *writer)

  Narrow the given ***DDSDataWriter*** *(p. 1113) pointer to a* ***DDSKeyedOctetsDataWriter*** *(p. 1276) pointer.*

6.194.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataWriter` < **DDS_KeyedOctets** (p. 765) >.

See also:

FooDataWriter (p. 1475)
DDSDataWriter (p. 1113)

6.194.2 Member Function Documentation

6.194.2.1 static DDSKeyedOctetsDataWriter* DDSKeyedOctetsDataWriter::narrow (DDSDataWriter * *writer*) [static]

Narrow the given **DDSDataWriter** (p. 1113) pointer to a **DDSKeyedOctetsDataWriter** (p. 1276) pointer.

See also:

FooDataWriter::narrow (p. 1477)

6.194.2.2 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataWriter::register_instance (const DDS_KeyedOctets & *instance_data*) [virtual]

Informs RTI Connext that the application will be modifying a particular instance.

See also:

FooDataWriter::register_instance (p. 1478)

6.194.2.3 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataWriter::register_instance (const char * *key*) [virtual]

<<*eXtension*>> *(p. 199)* Informs RTI Connext that the application will be modifying a particular instance.

See also:

FooDataWriter::register_instance (p. 1478)

6.194.2.4 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataWriter::register_instance_w_timestamp (const DDS_KeyedOctets & *instance_data*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same functions as **DDSKeyedOctetsDataWriter::register_instance** (p. 1280) except that the application provides the value for the `source_timestamp`.

See also:

FooDataWriter::register_instance_w_timestamp (p. 1479)

6.194.2.5 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataWriter::register_instance_w_timestamp (const char * *key*, const DDS_Time_t & *source_timestamp*) [virtual]

<<*eXtension*>> *(p. 199)* Performs the same functions as **DDSKeyedOctetsDataWriter::register_instance** (p. 1280) except that the application provides the value for the `source_timestamp`.

See also:

FooDataWriter::register_instance_w_timestamp (p. 1479)

6.194.2.6 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::unregister_instance (const DDS_KeyedOctets & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Reverses the action of **DDSKeyedOctetsDataWriter::register_instance** (p. 1280).

See also:

FooDataWriter::unregister_instance (p. 1480)

6.194.2.7 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::unregister_instance (const char * *key*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<*eXtension*>> *(p. 199)* Reverses the action of **DDSKeyedOctetsDataWriter::register_instance** (p. 1280).

See also:

FooDataWriter::unregister_instance (p. 1480)

6.194.2.8 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::unregister_instance_w_timestamp (const DDS_KeyedOctets & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::unregister_instance** (p. 1281) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::unregister_instance_w_timestamp (p. 1482)

6.194.2.9 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::unregister_instance_w_timestamp (const char * *key*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::unregister_instance** (p. 1281) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::unregister_instance_w_timestamp (p. 1482)

6.194.2.10 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write (const DDS_KeyedOctets & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Modifies the value of a **DDS_KeyedOctets** (p. 765) data instance.

See also:

FooDataWriter::write (p. 1484)

6.194.2.11 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write (const char * *key*, const unsigned char * *octets*, int *length*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<*eXtension*>> *(p. 199)* Modifies the value of a **DDS_KeyedOctets** (p. 765) data instance.

Parameters:

key *<<in>>* *(p. 200)* Instance key.

octets *<<in>>* *(p. 200)* Array of octets to be published.

length *<<in>>* *(p. 200)* Number of octets to be published.

handle *<<in>>* *(p. 200)* Either the handle returned by a previous call to **DDSKeyedOctetsDataWriter::register_instance** (p. 1280), or else the special value **DDS_HANDLE_NIL** (p. 55). See **FooDataWriter::write** (p. 1484).

See also:

FooDataWriter::write (p. 1484)

6.194.2.12 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write (const char * *key*, const DDS_OctetSeq & *octets*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<eXtension>> *(p. 199)* Modifies the value of a **DDS_KeyedOctets** (p. 765) data instance.

Parameters:

key *<<in>>* *(p. 200)* Instance key.

octets *<<in>>* *(p. 200)* Sequence of octets to be published.

handle *<<in>>* *(p. 200)* Either the handle returned by a previous call to **DDSKeyedOctetsDataWriter::register_instance** (p. 1280), or else the special value **DDS_HANDLE_NIL** (p. 55). See **FooDataWriter::write** (p. 1484).

See also:

FooDataWriter::write (p. 1484)

6.194.2.13 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write_w_timestamp (const DDS_KeyedOctets & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::write** (p. 1282) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.194.2.14 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write_w_timestamp (const char * *key*, const unsigned char * *octets*, int *length*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::write** (p. 1282) except that it also provides the value for the `source_timestamp`.

Parameters:

key *<<in>>* *(p. 200)* Instance key.

octets *<<in>>* *(p. 200)* Array of octets to be published.

length *<<in>>* *(p. 200)* Number of octets to be published.

handle *<<in>>* *(p. 200)* Either the handle returned by a previous call to **DDSKeyedOctetsDataWriter::register_instance** (p. 1280), or else the special value **DDS_HANDLE_NIL** (p. 55). See **FooDataWriter::write** (p. 1484).

source_timestamp *<<in>>* *(p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation. See **FooDataWriter::write_w_timestamp** (p. 1486).

See also:

FooDataWriter::write (p. 1484)

6.194.2.15 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write_w_timestamp (const char * *key*, const DDS_OctetSeq & *octets*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::write** (p. 1282) except that it also provides the value for the `source_timestamp`.

Parameters:

key *<<in>>* *(p. 200)* Instance key.

octets *<<in>>* *(p. 200)* Sequence of octets to be published.

handle *<<in>>* *(p. 200)* Either the handle returned by a previous call to **DDSKeyedOctetsDataWriter::register_instance** (p. 1280), or else the special value **DDS_HANDLE_NIL** (p. 55). See **FooDataWriter::write** (p. 1484).

source_timestamp *<<in>>* *(p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation. See **FooDataWriter::write_w_timestamp** (p. 1486).

See also:

FooDataWriter::write (p. 1484)

6.194.2.16 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write_w_params (const DDS_KeyedOctets & *instance_data*, DDS_WriteParams_t & *params*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::write** (p. 1282) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

See also:

FooDataWriter::write_w_params (p. 1487)

6.194.2.17 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write_w_params (const char * *key*, const unsigned char * *octets*, int *length*, DDS_WriteParams_t & *params*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::write** (p. 1282) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

Parameters:

key *<<in>>* *(p. 200)* Instance key.

octets *<<in>>* *(p. 200)* Array of octets to be published.

length *<<in>>* *(p. 200)* Number of octets to be published.

params *<<in>>* *(p. 200)* The **DDS_WriteParams_t** (p. 1067) parameter containing the instance handle, source timestamp, publication priority, and cookie to be used in write operation. See **FooDataWriter::write_w_params** (p. 1487).

See also:

FooDataWriter::write_w_params (p. 1487)

6.194.2.18 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::write_w_params (const char * *key*, const DDS_OctetSeq & *octets*, DDS_WriteParams_t & *params*) [virtual]

Performs the same function as **DDSKeyedOctetsDataWriter::write** (p. 1282) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

Parameters:

key *<<in>> (p. 200)* Instance key.

octets *<<in>> (p. 200)* Sequence of octets to be published.

params *<<in>> (p. 200)* The **DDS_WriteParams_t** (p. 1067) parameter containing the instance handle, source timestamp, publication priority, and cookie to be used in write operation. See **FooDataWriter::write_w_params** (p. 1487).

See also:

FooDataWriter::write_w_params (p. 1487)

6.194.2.19 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::dispose (const DDS_KeyedOctets & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Requests the middleware to delete the data.

See also:

FooDataWriter::dispose (p. 1489)

6.194.2.20 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::dispose (const char * *key*, const DDS_InstanceHandle_t & *instance_handle*) [virtual]

<<eXtension>> (p. 199) Requests the middleware to delete the data.

See also:

FooDataWriter::dispose (p. 1489)

6.194.2.21 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::dispose_w_timestamp (const DDS_KeyedOctets & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same functions as **DDSKeyedOctetsDataWriter::dispose** (p. 1286) except that the application provides the value for the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

See also:

FooDataWriter::dispose_w_timestamp (p. 1490)

6.194.2.22 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::dispose_w_timestamp (const char * *key*, const DDS_InstanceHandle_t & *instance_handle*, const DDS_Time_t & *source_timestamp*) [virtual]

<<*eXtension*>> *(p. 199)* Performs the same functions as **DDSKeyedOctetsDataWriter::dispose** (p. 1286) except that the application provides the value for the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

See also:

FooDataWriter::dispose_w_timestamp (p. 1490)

6.194.2.23 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::get_key_value (DDS_KeyedOctets & *key_holder*, const DDS_InstanceHandle_t & *handle*) [virtual]

Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataWriter::get_key_value (p. 1492)

6.194.2.24 virtual DDS_ReturnCode_t DDSKeyedOctetsDataWriter::get_key_value (char * *key*, const DDS_InstanceHandle_t & *handle*) [virtual]

*<<**eXtension**>>* *(p. 199)* Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataWriter::get_key_value (p. 1492)

6.194.2.25 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataWriter::lookup_instance (const DDS_KeyedOctets & *key_holder*) [virtual]

Retrieve the instance `handle` that corresponds to an instance `key_holder`.

See also:

FooDataWriter::lookup_instance (p. 1492)

6.194.2.26 virtual DDS_InstanceHandle_t DDSKeyedOctetsDataWriter::lookup_instance (const char * *key*) [virtual]

*<<**eXtension**>>* *(p. 199)* Retrieve the instance `handle` that corresponds to an instance `key`.

See also:

FooDataWriter::lookup_instance (p. 1492)

6.195 DDSKeyedOctetsTypeSupport Class Reference

<<*interface*>> *(p. 199)* **DDS_KeyedOctets** (p. 765) type support.

Inheritance diagram for DDSKeyedOctetsTypeSupport::

Static Public Member Functions

- static **DDS_ReturnCode_t register_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::KeyedOctets")

  Allows an application to communicate to RTI Connext the existence of the ***DDS_KeyedOctets*** *(p. 765) data type.*

- static **DDS_ReturnCode_t unregister_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::KeyedOctets")

  Allows an application to unregister the ***DDS_KeyedOctets*** *(p. 765) data type from RTI Connext. After calling unregister_type, no further communication using this type is possible.*

- static const char * **get_type_name** ()

  Get the default name for the ***DDS_KeyedOctets*** *(p. 765) type.*

- static void **print_data** (const **DDS_KeyedOctets** *a_data)

  <<**eXtension**>> (p. 199) *Print value of data type to standard out.*

6.195.1 Detailed Description

<<*interface*>> *(p. 199)* **DDS_KeyedOctets** (p. 765) type support.

6.195.2 Member Function Documentation

6.195.2.1 static DDS_ReturnCode_t DDSKeyedOctetsTypeSupport::register_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::KeyedOctets") [static]

Allows an application to communicate to RTI Connext the existence of the **DDS_KeyedOctets** (p. 765) data type.

By default, The **DDS_KeyedOctets** (p. 765) built-in type is automatically registered when a DomainParticipant is created using the type_name returned by **DDSKeyedOctetsTypeSupport::get_type_name** (p. 1291). Therefore, the usage of this function is optional and it is only required when the automatic built-in type registration is disabled using the participant property "dds.builtin_type.auto_register".

This method can also be used to register the same **DDSKeyedOctetsTypeSupport** (p. 1289) with a **DDSDomainParticipant** (p. 1139) using different values for the type_name.

If `register_type` is called multiple times with the same **DDSDomainParticipant** (p. 1139) and `type_name`, the second (and subsequent) registrations are ignored by the operation.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to register the data type **DDS_Octets** (p. 799) with. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type **DDS_KeyedOctets** (p. 765) is registered with the participant; this type name is used when creating a new **DDSTopic** (p. 1419). (See **DDSDomainParticipant::create_topic** (p. 1175).) The name may not be NULL or longer than 255 characters.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to register a type. Subsequent calls are thread safe.

See also:

DDSDomainParticipant::create_topic (p. 1175)

6.195.2.2 static DDS_ReturnCode_t DDSKeyedOctetsTypeSupport::unregister_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::KeyedOctets") [static]

Allows an application to unregister the **DDS_KeyedOctets** (p. 765) data type from RTI Connext. After calling unregister_type, no further communication

using this type is possible.

Precondition:

The **DDS_KeyedOctets** (p. 765) type with `type_name` is registered with the participant and all **DDSTopic** (p. 1419) objects referencing the type have been destroyed. If the type is not registered with the participant, or if any **DDSTopic** (p. 1419) is associated with the type, the operation will fail with **DDS_RETCODE_ERROR** (p. 315).

Postcondition:

All information about the type is removed from RTI Connext. No further communication using this type is possible.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to unregister the data type **DDS_KeyedOctets** (p. 765) from. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type **DDS_KeyedOctets** (p. 765) is registered with the participant. The name should match a name that has been previously used to register a type with the participant. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_BAD_PARAMETER** (p. 315) or **DDS_RETCODE_ERROR** (p. 315)

MT Safety:

SAFE.

See also:

DDSKeyedOctetsTypeSupport::register_type (p. 1289)

6.195.2.3 static const char* DDSKeyedOctetsTypeSupport::get_type_name () [static]

Get the default name for the **DDS_KeyedOctets** (p. 765) type.

Can be used for calling **DDSKeyedOctetsTypeSupport::register_type** (p. 1289) or creating **DDSTopic** (p. 1419).

Returns:

default name for the **DDS_KeyedOctets** (p. 765) type.

See also:

DDSKeyedOctetsTypeSupport::register_type (p. 1289)
DDSDomainParticipant::create_topic (p. 1175)

6.195.2.4 static void DDSKeyedOctetsTypeSupport::print_data (const DDS_KeyedOctets * *a_data*) [static]

*<<**eXtension**>> (p. 199)* Print value of data type to standard out.

The *generated* implementation of the operation knows how to print value of a data type.

Parameters:

a_data *<<**in**>> (p. 200)* **DDS_KeyedOctets** (p. 765) to be printed.

6.196 DDSKeyedStringDataReader Class Reference

<<*interface*>> *(p. 199)* Instantiates `DataReader` < **DDS_KeyedString** (p. 768) >.

Inheritance diagram for DDSKeyedStringDataReader::

Public Member Functions

- virtual **DDS_ReturnCode_t** **read** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  Access a collection of data samples from the ***DDSDataReader*** *(p. 1087).*

- virtual **DDS_ReturnCode_t** **take** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  Access a collection of data-samples from the ***DDSDataReader*** *(p. 1087).*

- virtual **DDS_ReturnCode_t** **read_w_condition** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  Accesses via ***DDSKeyedStringDataReader::read*** *(p. 1297) the samples that match the criteria specified in the* ***DDSReadCondition*** *(p. 1374).*

- virtual **DDS_ReturnCode_t** **take_w_condition** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  Analogous to ***DDSKeyedStringDataReader::read_w_condition*** *(p. 1298) except it accesses samples via the* ***DDSKeyedStringDataReader::take*** *(p. 1297) operation.*

- virtual **DDS_ReturnCode_t read_next_sample** (**DDS_KeyedString** &received_data, **DDS_SampleInfo** &sample_info)

  *Copies the next not-previously-accessed data value from the **DDS-DataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t take_next_sample** (**DDS_KeyedString** &received_data, **DDS_SampleInfo** &sample_info)

  *Copies the next not-previously-accessed data value from the **DDS-DataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t read_instance** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t take_instance** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t read_instance_w_condition** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

  *Accesses via **DDSKeyedStringDataReader::read_instance** (p. 1299) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).*

- virtual **DDS_ReturnCode_t take_instance_w_condition** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &a_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

*Accesses via **DDSKeyedStringDataReader::take_instance*** (p. 1299) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t** **read_next_instance** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader*** (p. 1087).

- virtual **DDS_ReturnCode_t** **take_next_instance** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  *Access a collection of data samples from the **DDSDataReader*** (p. 1087).

- virtual **DDS_ReturnCode_t** **read_next_instance_w_condition** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

  *Accesses via **DDSKeyedStringDataReader::read_next_instance*** (p. 1300) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t** **take_next_instance_w_condition** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, const **DDS_InstanceHandle_t** &previous_handle=**DDS_HANDLE_NIL**, **DDSReadCondition** *condition=NULL)

  *Accesses via **DDSKeyedStringDataReader::take_next_instance*** (p. 1301) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t** **return_loan** (**DDS_KeyedStringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq)

*Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of* `received_data` *and* `info_seq` *obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t get_key_value** (**DDS_KeyedString** &key_holder, const **DDS_InstanceHandle_t** &handle)

  Retrieve the instance `key` *that corresponds to an instance* `handle`*.*

- virtual **DDS_ReturnCode_t get_key_value** (char *key, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `key` *that corresponds to an instance* `handle`*.*

- virtual **DDS_InstanceHandle_t lookup_instance** (const **DDS_KeyedString** &key_holder)

  Retrieve the instance `handle` *that corresponds to an instance* `key_holder`*.*

- virtual **DDS_InstanceHandle_t lookup_instance** (const char *key)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `handle` *that corresponds to an instance* `key`*.*

Static Public Member Functions

- static **DDSKeyedStringDataReader** * **narrow** (**DDSDataReader** *reader)

  *Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSKeyedStringDataReader** (p. 1293) pointer.*

6.196.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataReader` < **DDS_KeyedString** (p. 768) >.

When reading or taking data with this reader, if you request a copy of the samples instead of a loan, and a string in a destination data sample is NULL, the middleware will allocate a new string for you of sufficient length to hold the received string. The new string will be allocated with **DDS_String_alloc** (p. 458); the sample's destructor will delete it.

A non- NULL string is assumed to be allocated to sufficient length to store the incoming data. It will not be reallocated.

See also:

FooDataReader (p. 1444)
DDSDataReader (p. 1087)

6.196.2 Member Function Documentation

6.196.2.1 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read (p. 1447)

6.196.2.2 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data-samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take (p. 1448)

6.196.2.3 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read_w_condition (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Accesses via **DDSKeyedStringDataReader::read** (p. 1297) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_w_condition (p. 1454)

6.196.2.4 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take_w_condition (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Analogous to **DDSKeyedStringDataReader::read_w_condition** (p. 1298) except it accesses samples via the **DDSKeyedStringDataReader::take** (p. 1297) operation.

See also:

FooDataReader::take_w_condition (p. 1456)

6.196.2.5 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read_next_sample (DDS_KeyedString & *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_next_sample (p. 1457)

6.196.2.6 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take_next_sample (DDS_KeyedString & *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_next_sample (p. 1458)

6.196.2.7 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read_instance (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_instance (p. 1459)

6.196.2.8 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take_instance (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_instance (p. 1460)

6.196.2.9 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read_instance_w_condition (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedStringDataReader::read_instance** (p. 1299) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_instance_w_condition (p. 1462)

6.196.2.10 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take_instance_w_condition (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *a_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedStringDataReader::take_instance** (p. 1299) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::take_instance_w_condition (p. 1463)

6.196.2.11 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read_next_instance (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_next_instance (p. 1464)

6.196.2.12 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take_next_instance (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_next_instance (p. 1467)

6.196.2.13 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::read_next_instance_w_condition (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedStringDataReader::read_next_instance** (p. 1300) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_next_instance_w_condition (p. 1468)

6.196.2.14 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::take_next_instance_w_condition (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, const DDS_InstanceHandle_t & *previous_handle* = DDS_HANDLE_NIL, DDSReadCondition * *condition* = NULL) [virtual]

Accesses via **DDSKeyedStringDataReader::take_next_instance** (p. 1301) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::take_next_instance_w_condition (p. 1470)

6.196.2.15 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::return_loan (DDS_KeyedStringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*) [virtual]

Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of `received_data` and `info_seq` obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).

See also:

FooDataReader::return_loan (p. 1471)

6.196.2.16 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::get_key_value (DDS_KeyedString & *key_holder*, const DDS_InstanceHandle_t & *handle*) [virtual]

Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataReader::get_key_value (p. 1472)

6.196.2.17 virtual DDS_ReturnCode_t DDSKeyedStringDataReader::get_key_value (char * *key*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<*eXtension*>> *(p. 199)* Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataReader::get_key_value (p. 1472)

6.196.2.18 virtual DDS_InstanceHandle_t DDSKeyedStringDataReader::lookup_instance (const DDS_KeyedString & *key_holder*) [virtual]

Retrieve the instance `handle` that corresponds to an instance `key_holder`.

See also:

FooDataReader::lookup_instance (p. 1473)

6.196.2.19 virtual DDS_InstanceHandle_t DDSKeyedStringDataReader::lookup_instance (const char * *key*) [virtual]

<<*eXtension*>> *(p. 199)* Retrieve the instance `handle` that corresponds to an instance `key`.

See also:

FooDataReader::lookup_instance (p. 1473)

6.196.2.20 static DDSKeyedStringDataReader* DDSKeyedStringDataReader::narrow (DDSDataReader * *reader*) [static]

Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSKeyedStringDataReader** (p. 1293) pointer.

See also:

FooDataReader::narrow (p. 1447)

6.197 DDSKeyedStringDataWriter Class Reference

<<***interface***>> *(p. 199)* Instantiates `DataWriter` < **DDS_KeyedString** (p. 768) >.

Inheritance diagram for DDSKeyedStringDataWriter::

Public Member Functions

- virtual **DDS_InstanceHandle_t register_instance** (const **DDS_KeyedString** &instance_data)

  Informs RTI Connext that the application will be modifying a particular instance.

- virtual **DDS_InstanceHandle_t register_instance** (const char *key)

  <<**eXtension**>> (p. 199) *Informs RTI Connext that the application will be modifying a particular instance.*

- virtual **DDS_InstanceHandle_t register_instance_w_timestamp** (const **DDS_KeyedString** &instance_data, const **DDS_Time_t** &source_timestamp)

  Performs the same functions as ***DDSKeyedStringDataWriter::register_instance*** *(p. 1307) except that the application provides the value for the* `source_timestamp`*.*

- virtual **DDS_InstanceHandle_t register_instance_w_timestamp** (const char *key, const **DDS_Time_t** &source_timestamp)

  <<**eXtension**>> (p. 199) *Performs the same functions as* ***DDSKeyedStringDataWriter::register_instance*** *(p. 1307) except that the application provides the value for the* `source_timestamp`*.*

- virtual **DDS_ReturnCode_t unregister_instance** (const **DDS_KeyedString** &instance_data, const **DDS_InstanceHandle_t** &handle)

  Reverses the action of ***DDSKeyedStringDataWriter::register_instance*** *(p. 1307).*

- virtual **DDS_ReturnCode_t unregister_instance** (const char *key, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Reverses the action of* ***DDSKeyedStringDataWriter::register_instance*** *(p. 1307).*

^ virtual **DDS_ReturnCode_t unregister_instance_w_timestamp** (const **DDS_KeyedString** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

*Performs the same function as **DDSKeyedStringDataWriter::unregister_instance** (p. 1309) except that it also provides the value for the* `source_timestamp`.

^ virtual **DDS_ReturnCode_t unregister_instance_w_timestamp** (const char *key, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

<<**eXtension**>> (p. 199) *Performs the same function as **DDSKeyedStringDataWriter::unregister_instance** (p. 1309) except that it also provides the value for the* `source_timestamp`.

^ virtual **DDS_ReturnCode_t write** (const **DDS_KeyedString** &instance_data, const **DDS_InstanceHandle_t** &handle)

*Modifies the value of a **DDS_KeyedString** (p. 768) data instance.*

^ virtual **DDS_ReturnCode_t write** (const char *key, const char *str, const **DDS_InstanceHandle_t** &handle)

<<**eXtension**>> (p. 199) *Modifies the value of a **DDS_KeyedString** (p. 768) data instance.*

^ virtual **DDS_ReturnCode_t write_w_timestamp** (const **DDS_KeyedString** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

*Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also provides the value for the* `source_timestamp`.

^ virtual **DDS_ReturnCode_t write_w_timestamp** (const char *key, const char *str, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

<<**eXtension**>> (p. 199) *Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also provides the value for the* `source_timestamp`.

^ virtual **DDS_ReturnCode_t write_w_params** (const **DDS_KeyedString** &instance_data, **DDS_WriteParams_t** &params)

*Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

^ virtual **DDS_ReturnCode_t write_w_params** (const char *key, const char *str, **DDS_WriteParams_t** &params)

<<**eXtension**>> (p. 199) *Performs the same function as* ***DDSKeyedStringDataWriter::write*** *(p. 1310) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

- virtual **DDS_ReturnCode_t dispose** (const **DDS_KeyedString** &instance_data, const **DDS_InstanceHandle_t** &handle)

  Requests the middleware to delete the data.

- virtual **DDS_ReturnCode_t dispose** (const char *key, const **DDS_InstanceHandle_t** &instance_handle)

  <<**eXtension**>> (p. 199) *Requests the middleware to delete the data.*

- virtual **DDS_ReturnCode_t dispose_w_timestamp** (const **DDS_KeyedString** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

  Performs the same functions as ***DDSKeyedStringDataWriter::dispose*** *(p. 1311) except that the application provides the value for the* `source_timestamp` *that is made available to* ***DDSDataReader*** *(p. 1087) objects by means of the* `source_timestamp` *attribute inside the* ***DDS_SampleInfo*** *(p. 912).*

- virtual **DDS_ReturnCode_t dispose_w_timestamp** (const char *key, const **DDS_InstanceHandle_t** &instance_handle, const **DDS_Time_t** &source_timestamp)

  <<**eXtension**>> (p. 199) *Performs the same functions as* ***DDSKeyedStringDataWriter::dispose*** *(p. 1311) except that the application provides the value for the* `source_timestamp` *that is made available to* ***DDSDataReader*** *(p. 1087) objects by means of the* `source_timestamp` *attribute inside the* ***DDS_SampleInfo*** *(p. 912).*

- virtual **DDS_ReturnCode_t get_key_value** (**DDS_KeyedString** &key_holder, const **DDS_InstanceHandle_t** &handle)

  Retrieve the instance `key` *that corresponds to an instance* `handle`*.*

- virtual **DDS_ReturnCode_t get_key_value** (char *key, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Retrieve the instance* `key` *that corresponds to an instance* `handle`*.*

- virtual **DDS_InstanceHandle_t lookup_instance** (const **DDS_KeyedString** &key_holder)

  Retrieve the instance `handle` *that corresponds to an instance* `key_holder`*.*

- virtual **DDS_InstanceHandle_t lookup_instance** (const char *key)

<<**eXtension**>> (p. 199) *Retrieve the instance* `handle` *that corresponds to an instance* `key`.

Static Public Member Functions

- static **DDSKeyedStringDataWriter** * **narrow** (**DDSDataWriter** *writer)

  Narrow the given ***DDSDataWriter*** *(p. 1113) pointer to a* ***DDSKeyedStringDataWriter*** *(p. 1304) pointer.*

6.197.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataWriter` < **DDS_KeyedString** (p. 768) >.

See also:

FooDataWriter (p. 1475)
DDSDataWriter (p. 1113)

6.197.2 Member Function Documentation

6.197.2.1 static DDSKeyedStringDataWriter* DDSKeyedStringDataWriter::narrow (DDSDataWriter * *writer*) [static]

Narrow the given **DDSDataWriter** (p. 1113) pointer to a **DDSKeyedStringDataWriter** (p. 1304) pointer.

See also:

FooDataWriter::narrow (p. 1477)

6.197.2.2 virtual DDS_InstanceHandle_t DDSKeyedStringDataWriter::register_instance (const DDS_KeyedString & *instance_data*) [virtual]

Informs RTI Connext that the application will be modifying a particular instance.

See also:

FooDataWriter::register_instance (p. 1478)

6.197.2.3 virtual DDS_InstanceHandle_t DDSKeyedStringDataWriter::register_instance (const char * *key*) [virtual]

<<*eXtension*>> *(p. 199)* Informs RTI Connext that the application will be modifying a particular instance.

See also:

FooDataWriter::register_instance (p. 1478)

6.197.2.4 virtual DDS_InstanceHandle_t DDSKeyedStringDataWriter::register_instance_w_timestamp (const DDS_KeyedString & *instance_data*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same functions as **DDSKeyedStringDataWriter::register_instance** (p. 1307) except that the application provides the value for the `source_timestamp`.

See also:

FooDataWriter::register_instance_w_timestamp (p. 1479)

6.197.2.5 virtual DDS_InstanceHandle_t DDSKeyedStringDataWriter::register_instance_w_timestamp (const char * *key*, const DDS_Time_t & *source_timestamp*) [virtual]

<<*eXtension*>> *(p. 199)* Performs the same functions as **DDSKeyedStringDataWriter::register_instance** (p. 1307) except that the application provides the value for the `source_timestamp`.

See also:

FooDataWriter::register_instance_w_timestamp (p. 1479)

6.197.2.6 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::unregister_instance (const DDS_KeyedString & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Reverses the action of **DDSKeyedStringDataWriter::register_instance** (p. 1307).

See also:

FooDataWriter::unregister_instance (p. 1480)

6.197.2.7 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::unregister_instance (const char * *key*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<*eXtension*>> *(p. 199)* Reverses the action of **DDSKeyedStringDataWriter::register_instance** (p. 1307).

See also:

FooDataWriter::unregister_instance (p. 1480)

6.197.2.8 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::unregister_instance_w_timestamp (const DDS_KeyedString & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedStringDataWriter::unregister_instance** (p. 1309) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::unregister_instance_w_timestamp (p. 1482)

6.197.2.9 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::unregister_instance_w_timestamp (const char * *key*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

<<*eXtension*>> *(p. 199)* Performs the same function as **DDSKeyedStringDataWriter::unregister_instance** (p. 1309) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::unregister_instance_w_timestamp (p. 1482)

6.197.2.10 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::write (const DDS_KeyedString & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Modifies the value of a **DDS_KeyedString** (p. 768) data instance.

See also:

FooDataWriter::write (p. 1484)

6.197.2.11 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::write (const char * *key*, const char * *str*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<eXtension>> *(p. 199)* Modifies the value of a **DDS_KeyedString** (p. 768) data instance.

See also:

FooDataWriter::write (p. 1484)

6.197.2.12 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::write_w_timestamp (const DDS_KeyedString & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.197.2.13 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::write_w_timestamp (const char * *key*, const char * *str*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

<<eXtension>> (p. 199) Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.197.2.14 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::write_w_params (const DDS_KeyedString & *instance_data*, DDS_WriteParams_t & *params*) [virtual]

Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

See also:

FooDataWriter::write_w_params (p. 1487)

6.197.2.15 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::write_w_params (const char * *key*, const char * *str*, DDS_WriteParams_t & *params*) [virtual]

<<eXtension>> (p. 199) Performs the same function as **DDSKeyedStringDataWriter::write** (p. 1310) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

See also:

FooDataWriter::write_w_params (p. 1487)

6.197.2.16 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::dispose (const DDS_KeyedString & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Requests the middleware to delete the data.

See also:

FooDataWriter::dispose (p. 1489)

6.197.2.17 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::dispose (const char * *key*, const DDS_InstanceHandle_t & *instance_handle*) [virtual]

<<*eXtension*>> *(p. 199)* Requests the middleware to delete the data.

See also:

FooDataWriter::dispose (p. 1489)

6.197.2.18 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::dispose_w_timestamp (const DDS_KeyedString & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same functions as **DDSKeyedStringDataWriter::dispose** (p. 1311) except that the application provides the value for the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

See also:

FooDataWriter::dispose_w_timestamp (p. 1490)

6.197.2.19 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::dispose_w_timestamp (const char * *key*, const DDS_InstanceHandle_t & *instance_handle*, const DDS_Time_t & *source_timestamp*) [virtual]

<<*eXtension*>> *(p. 199)* Performs the same functions as **DDSKeyedStringDataWriter::dispose** (p. 1311) except that the application provides the value for the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

See also:

FooDataWriter::dispose_w_timestamp (p. 1490)

6.197.2.20 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::get_key_value (DDS_KeyedString & *key_holder*, const DDS_InstanceHandle_t & *handle*) [virtual]

Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataWriter::get_key_value (p. 1492)

6.197.2.21 virtual DDS_ReturnCode_t DDSKeyedStringDataWriter::get_key_value (char * *key*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<*eXtension*>> *(p. 199)* Retrieve the instance `key` that corresponds to an instance `handle`.

See also:

FooDataWriter::get_key_value (p. 1492)

6.197.2.22 virtual DDS_InstanceHandle_t DDSKeyedStringDataWriter::lookup_instance (const DDS_KeyedString & *key_holder*) [virtual]

Retrieve the instance `handle` that corresponds to an instance `key_holder`.

See also:

FooDataWriter::lookup_instance (p. 1492)

6.197.2.23 virtual DDS_InstanceHandle_t DDSKeyedStringDataWriter::lookup_instance (const char * *key*) [virtual]

<<*eXtension*>> *(p. 199)* Retrieve the instance `handle` that corresponds to an instance `key`.

See also:

FooDataWriter::lookup_instance (p. 1492)

6.198 DDSKeyedStringTypeSupport Class Reference

<<*interface*>> *(p. 199)* Keyed string type support.

Inheritance diagram for DDSKeyedStringTypeSupport::

Static Public Member Functions

- static **DDS_ReturnCode_t register_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::KeyedString")

  Allows an application to communicate to RTI Connext the existence of the ***DDS_KeyedString*** *(p. 768) data type.*

- static **DDS_ReturnCode_t unregister_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::KeyedString")

  Allows an application to unregister the ***DDS_KeyedString*** *(p. 768) data type from RTI Connext. After calling unregister_type, no further communication using this type is possible.*

- static const char * **get_type_name** ()

  Get the default name for the ***DDS_KeyedString*** *(p. 768) type.*

- static void **print_data** (const **DDS_KeyedString** *a_data)

  <<**eXtension**>> (p. 199) *Print value of data type to standard out.*

6.198.1 Detailed Description

<<*interface*>> *(p. 199)* Keyed string type support.

6.198.2 Member Function Documentation

6.198.2.1 static DDS_ReturnCode_t DDSKeyedStringTypeSupport::register_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::KeyedString") [static]

Allows an application to communicate to RTI Connext the existence of the **DDS_KeyedString** (p. 768) data type.

By default, The **DDS_KeyedString** (p. 768) built-in type is automatically registered when a DomainParticipant is created using the type_name returned by **DDSKeyedStringTypeSupport::get_type_name** (p. 1316). Therefore, the usage of this function is optional and it is only required when the automatic built-in type registration is disabled using the participant property "dds.builtin_type.auto_register".

This method can also be used to register the same **DDSKeyedStringTypeSupport** (p. 1314) with a **DDSDomainParticipant** (p. 1139) using different values for the type_name.

If `register_type` is called multiple times with the same **DDSDomainParticipant** (p. 1139) and `type_name`, the second (and subsequent) registrations are ignored by the operation.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to register the data type **DDS_KeyedString** (p. 768) with. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type **DDS_KeyedString** (p. 768) is registered with the participant; this type name is used when creating a new **DDSTopic** (p. 1419). (See **DDSDomainParticipant::create_topic** (p. 1175).) The name may not be NULL or longer than 255 characters.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to register a type. Subsequent calls are thread safe.

See also:

DDSDomainParticipant::create_topic (p. 1175)

6.198.2.2 static DDS_ReturnCode_t DDSKeyedStringTypeSupport::unregister_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::KeyedString") [static]

Allows an application to unregister the **DDS_KeyedString** (p. 768) data type from RTI Connext. After calling unregister_type, no further communication

using this type is possible.

Precondition:

The **DDS_KeyedString** (p. 768) type with `type_name` is registered with the participant and all **DDSTopic** (p. 1419) objects referencing the type have been destroyed. If the type is not registered with the participant, or if any **DDSTopic** (p. 1419) is associated with the type, the operation will fail with **DDS_RETCODE_ERROR** (p. 315).

Postcondition:

All information about the type is removed from RTI Connext. No further communication using this type is possible.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to unregister the data type **DDS_KeyedString** (p. 768) from. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type **DDS_KeyedString** (p. 768) is registered with the participant. The name should match a name that has been previously used to register a type with the participant. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_BAD_PARAMETER** (p. 315) or **DDS_RETCODE_ERROR** (p. 315)

MT Safety:

SAFE.

See also:

DDSKeyedStringTypeSupport::register_type (p. 1314)

6.198.2.3 static const char* DDSKeyedStringTypeSupport::get_type_name () [static]

Get the default name for the **DDS_KeyedString** (p. 768) type.

Can be used for calling **DDSKeyedStringTypeSupport::register_type** (p. 1314) or creating **DDSTopic** (p. 1419).

Returns:

default name for the **DDS_KeyedString** (p. 768) type.

See also:

DDSKeyedStringTypeSupport::register_type (p. 1314)
DDSDomainParticipant::create_topic (p. 1175)

6.198.2.4 static void DDSKeyedStringTypeSupport::print_data (const DDS_KeyedString * *a_data*) [static]

<<eXtension>> (p. 199) Print value of data type to standard out.

The *generated* implementation of the operation knows how to print value of a data type.

Parameters:

a_data *<<in>> (p. 200)* **DDS_KeyedString** (p. 768) to be printed.

6.199 DDSListener Class Reference

<<*interface*>> *(*p. *199)* Abstract base class for all Listener interfaces.

Inheritance diagram for DDSListener::

6.199.1 Detailed Description

<<*interface*>> *(*p. *199)* Abstract base class for all Listener interfaces.

Entity:

DDSEntity (p. 1253)

QoS:

QoS Policies (p. 331)

Status:

Status Kinds (p. 317)

All the supported kinds of concrete **DDSListener** (p. 1318) interfaces (one per concrete **DDSEntity** (p. 1253) type) derive from this root and add methods whose prototype depends on the concrete Listener.

Listeners provide a way for RTI Connext to asynchronously alert the application when there are relevant status changes.

Almost every application will have to implement listener interfaces.

Each dedicated listener presents a list of operations that correspond to the relevant communication status changes to which an application may respond.

The same **DDSListener** (p. 1318) instance may be shared among multiple entities if you so desire. Consequently, the provided parameter contains a reference to the concerned **DDSEntity** (p. 1253).

6.199.2 Access to Plain Communication Status

The general mapping between the plain communication statuses (see **Status Kinds** (p. 317)) and the listeners' operations is as follows:

- For each communication status, there is a corresponding operation whose name is `on_<communication_status>()`, which takes a parameter of type `<communication_status>` as listed in **Status Kinds** (p. 317).

- on_<communication_status> is available on the relevant **DDSEntity** (p. 1253) as well as those that embed it, as expressed in the following figure:

- When the application attaches a listener on an entity, it must set a mask. The mask indicates to RTI Connext which operations are enabled within the listener (cf. operation **DDSEntity** (p. 1253) set_listener()).

- When a plain communication status changes, RTI Connext triggers the most specific relevant listener operation that is enabled. In case the most specific relevant listener operation corresponds to an application-installed 'nil' listener the operation will be considered handled by a NO-OP operation that does not reset the communication status.

This behavior allows the application to set a default behavior (e.g., in the listener associated with the **DDSDomainParticipant** (p. 1139)) and to set dedicated behaviors only where needed.

6.199.3 Access to Read Communication Status

The two statuses related to data arrival are treated slightly differently. Since they constitute the core purpose of the Data Distribution Service, there is no need to provide a default mechanism (as is done for the plain communication statuses above).

The rule is as follows. Each time the read communication status changes:

- First, RTI Connext tries to trigger the **DDSSubscriberListener::on_data_on_readers** (p. 1415) with a parameter of the related **DDSSubscriber** (p. 1390);

- If this does not succeed (there is no listener or the operation is not enabled), RTI Connext tries to trigger **DDSDataReaderListener::on_data_available** (p. 1110) on all the related **DDSDataReaderListener** (p. 1108) objects, with a parameter of the related **DDSDataReader** (p. 1087).

The rationale is that either the application is interested in relations among data arrivals and it must use the first option (and then get the corresponding **DDSDataReader** (p. 1087) objects by calling **DDSSubscriber::get_datareaders** (p. 1407) on the related **DDSSubscriber** (p. 1390) and then get the data by calling **FooDataReader::read** (p. 1447) or **FooDataReader::take** (p. 1448) on the returned **DDSDataReader** (p. 1087) objects), or it wants to treat each

DDSDataReader (p. 1087) independently and it may choose the second option (and then get the data by calling **FooDataReader::read** (p. 1447) or **FooDataReader::take** (p. 1448) on the related **DDSDataReader** (p. 1087)).

Note that if **DDSSubscriberListener::on_data_on_readers** (p. 1415) is called, RTI Connext will *not* try to call **DDSDataReaderListener::on_data_available** (p. 1110). However, an application can force a call to the **DDSDataReader** (p. 1087) objects that have data by calling **DDSSubscriber::notify_datareaders** (p. 1408).

6.199.4 Operations Allowed in Listener Callbacks

The operations that are allowed in **DDSListener** (p. 1318) callbacks depend on the **DDS_ExclusiveAreaQosPolicy** (p. 742) QoS policy of the **DDSEntity** (p. 1253) to which the **DDSListener** (p. 1318) is attached – or in the case of a **DDSDataWriter** (p. 1113) of **DDSDataReader** (p. 1087) listener, on the **DDS_ExclusiveAreaQosPolicy** (p. 742) QoS of the parent **DDSPublisher** (p. 1346) or **DDSSubscriber** (p. 1390). For instance, the **DDS_ExclusiveAreaQosPolicy** (p. 742) settings of a **DDSSubscriber** (p. 1390) will determine which operations are allowed within the callbacks of the listeners associated with all the DataReaders created through that **DDSSubscriber** (p. 1390).

Note: these restrictions do not apply to builtin topic listener callbacks.

Regardless of whether **DDS_ExclusiveAreaQosPolicy::use_shared_exclusive_area** (p. 743) is set to **DDS_BOOLEAN_TRUE** (p. 298) or **DDS_BOOLEAN_FALSE** (p. 299), the following operations are *not* allowed:

- Within any listener callback, deleting the entity to which the **DDSListener** (p. 1318) is attached
- Within a **DDSTopic** (p. 1419) listener callback, any operations on any subscribers, readers, publishers or writers

An attempt to call a disallowed method from within a callback will result in **DDS_RETCODE_ILLEGAL_OPERATION** (p. 316).

If **DDS_ExclusiveAreaQosPolicy::use_shared_exclusive_area** (p. 743) is set to **DDS_BOOLEAN_FALSE** (p. 299), the setting which allows more concurrency among RTI Connext threads, the following are *not* allowed:

- Within any listener callback, creating any entity
- Within any listener callback, deleting any entity
- Within any listener callback, enabling any entity

- Within any listener callback, setting the QoS of any entities
- Within a **DDSDataReader** (p. 1087) or **DDSSubscriber** (p. 1390) listener callback, invoking any operation on any other **DDSSubscriber** (p. 1390) or on any **DDSDataReader** (p. 1087) belonging to another **DDSSubscriber** (p. 1390).
- Within a **DDSDataReader** (p. 1087) or **DDSSubscriber** (p. 1390) listener callback, invoking any operation on any **DDSPublisher** (p. 1346) (or on any **DDSDataWriter** (p. 1113) belonging to such a **DDSPublisher** (p. 1346)) that has **DDS_ExclusiveAreaQosPolicy::use_shared_exclusive_area** (p. 743) set to **DDS_BOOLEAN_TRUE** (p. 298).
- Within a **DDSDataWriter** (p. 1113) of **DDSPublisher** (p. 1346) listener callback, invoking any operation on another Publisher or on a **DDSDataWriter** (p. 1113) belonging to another **DDSPublisher** (p. 1346).
- Within a **DDSDataWriter** (p. 1113) of **DDSPublisher** (p. 1346) listener callback, invoking any operation on any **DDSSubscriber** (p. 1390) or **DDSDataReader** (p. 1087).

An attempt to call a disallowed method from within a callback will result in **DDS_RETCODE_ILLEGAL_OPERATION** (p. 316).

The above limitations can be lifted by setting **DDS_ExclusiveAreaQosPolicy::use_shared_exclusive_area** (p. 743) to **DDS_BOOLEAN_TRUE** (p. 298) on the **DDSPublisher** (p. 1346) or **DDSSubscriber** (p. 1390) (or on the **DDSPublisher** (p. 1346)/ **DDSSubscriber** (p. 1390) of the **DDSDataWriter** (p. 1113)/**DDSDataReader** (p. 1087)) to which the listener is attached. However, the application will pay the cost of reduced concurrency between the affected publishers and subscribers.

See also:

EXCLUSIVE_AREA (p. 430)
Status Kinds (p. 317)
DDSWaitSet (p. 1433), **DDSCondition** (p. 1075)

6.200 DDSMultiTopic Class Reference

[**Not supported (optional)**] <<***interface***>> *(*p. *199)* A specialization of **DDSTopicDescription** (p. 1427) that allows subscriptions that combine/filter/rearrange data coming from several topics.

Inheritance diagram for DDSMultiTopic::

Public Member Functions

- virtual const char * **get_subscription_expression** ()=0

  Get the expression for this ***DDSMultiTopic*** *(*p. *1322).*

- virtual **DDS_ReturnCode_t** **get_expression_parameters** (**DDS_StringSeq** &parameters)=0

  Get the expression parameters.

- virtual **DDS_ReturnCode_t** **set_expression_parameters** (const **DDS_StringSeq** &parameters)=0

  Set the `expression_parameters`.

Static Public Member Functions

- static **DDSMultiTopic** * **narrow** (**DDSTopicDescription** *topic_description)

  Narrow the given ***DDSTopicDescription*** *(*p. *1427) pointer to a* ***DDSMultiTopic*** *(*p. *1322) pointer.*

6.200.1 Detailed Description

[**Not supported (optional)**] <<***interface***>> *(*p. *199)* A specialization of **DDSTopicDescription** (p. 1427) that allows subscriptions that combine/filter/rearrange data coming from several topics.

DDSMultiTopic (p. 1322) allows a more sophisticated subscription that can select and combine data received from multiple topics into a single resulting type (specified by the inherited `type_name`). The data will then be filtered (selection) and possibly re-arranged (aggregation/projection) according to a `subscription_expression` with parameters `expression_parameters`.

- The `subscription_expression` is a string that identifies the selection and re-arrangement of data from the associated topics. It is similar to an SQL statement where the SELECT part provides the fields to be kept, the FROM part provides the names of the topics that are searched for those fields, and the WHERE clause gives the content filter. The Topics combined may have different types but they are restricted in that the type of the fields used for the NATURAL JOIN operation must be the same.

- The `expression_parameters` attribute is a sequence of strings that give values to the 'parameters' (i.e. "%n" tokens) in the `subscription_expression`. The number of supplied parameters must fit with the requested values in the `subscription_expression` (i.e. the number of n tokens).

- **DDSDataReader** (p. 1087) entities associated with a **DDSMultiTopic** (p. 1322) are alerted of data modifications by the usual **DDSListener** (p. 1318) or **DDSWaitSet** (p. 1433) / **DDSCondition** (p. 1075) mechanisms whenever modifications occur to the data associated with any of the topics relevant to the **DDSMultiTopic** (p. 1322).

Note that the source for data may not be restricted to a single topic.

DDSDataReader (p. 1087) entities associated with a **DDSMultiTopic** (p. 1322) may access instances that are "constructed" at the **DDSDataReader** (p. 1087) side from the instances written by multiple **DDSDataWriter** (p. 1113) entities. The **DDSMultiTopic** (p. 1322) access instance will begin to exist as soon as all the constituting **DDSTopic** (p. 1419) instances are in existence. The `view_state` and `instance_state` is computed from the corresponding states of the constituting instances:

- The `view_state` of the **DDSMultiTopic** (p. 1322) instance is `DDS_NEW_VIEW_STATE` (p. 114) if at least one of the constituting instances has `view_state` = **DDS_NEW_VIEW_STATE** (p. 114). Otherwise, it will be **DDS_NOT_NEW_VIEW_STATE** (p. 114).

- The `instance_state` of the **DDSMultiTopic** (p. 1322) instance is **DDS_ALIVE_INSTANCE_STATE** (p. 117) if the `instance_state` of all the constituting **DDSTopic** (p. 1419) instances is **DDS_ALIVE_INSTANCE_STATE** (p. 117). It is **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) if at least one of the constituting **DDSTopic** (p. 1419) instances is **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117). Otherwise, it is **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117).

Queries and Filters Syntax (p. 208) describes the syntax of `subscription_expression` and `expression_parameters`.

6.200.2 Member Function Documentation

6.200.2.1 static DDSMultiTopic* DDSMultiTopic::narrow (DDSTopicDescription * *topic_description*) [static]

Narrow the given **DDSTopicDescription** (p. 1427) pointer to a **DDSMultiTopic** (p. 1322) pointer.

Returns:

DDSMultiTopic (p. 1322) if this **DDSTopicDescription** (p. 1427) is a **DDSMultiTopic** (p. 1322). Otherwise, return NULL.

6.200.2.2 virtual const char* DDSMultiTopic::get_subscription_expression () [pure virtual]

Get the expression for this **DDSMultiTopic** (p. 1322).

The expressions syntax is described in the DDS specification. It is specified when the **DDSMultiTopic** (p. 1322) is created.

Returns:

`subscription_expression` of the **DDSMultiTopic** (p. 1322).

6.200.2.3 virtual DDS_ReturnCode_t DDSMultiTopic::get_expression_parameters (DDS_StringSeq & *parameters*) [pure virtual]

Get the expression parameters.

The expressions syntax is described in the DDS specification.

The `parameters` is either specified on the last successful call to **DDSMultiTopic::set_expression_parameters** (p. 1325), or if **DDSMultiTopic::set_expression_parameters** (p. 1325) was never called, the `parameters` specified when the **DDSMultiTopic** (p. 1322) was created.

Parameters:

parameters *<<inout>>* *(p. 200)* Fill in this sequence with the expression parameters. The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

Returns:

One of the **Standard Return Codes** (p. 314)

6.200.2.4 virtual DDS_ReturnCode_t DDSMultiTopic::set_expression_parameters (const DDS_StringSeq & *parameters*) [pure virtual]

Set the `expression_parameters`.

Changes the `expression_parameters` associated with the **DDSMultiTopic** (p. 1322).

Parameters:

parameters *<<in>> (p. 200)* the filter expression parameters

Returns:

One of the **Standard Return Codes** (p. 314).

6.201 DDSOctetsDataReader Class Reference

<<*interface*>> *(*p. *199)* Instantiates `DataReader` < **DDS_Octets** (p. 799) >.

Inheritance diagram for DDSOctetsDataReader::

Public Member Functions

- virtual **DDS_ReturnCode_t read** (**DDS_OctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  Access a collection of data samples from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t take** (**DDS_OctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  Access a collection of data-samples from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t read_w_condition** (**DDS_OctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  Accesses via ***DDSOctetsDataReader::read*** *(*p. *1328) the samples that match the criteria specified in the* ***DDSReadCondition*** *(*p. *1374).*

- virtual **DDS_ReturnCode_t take_w_condition** (**DDS_OctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  Analogous to ***DDSOctetsDataReader::read_w_condition*** *(*p. *1328) except it accesses samples via the* ***DDSOctetsDataReader::take*** *(*p. *1328) operation.*

- virtual **DDS_ReturnCode_t read_next_sample** (**DDS_Octets** &received_data, **DDS_SampleInfo** &sample_info)

  Copies the next not-previously-accessed data value from the ***DDSDataReader*** *(*p. *1087).*

^ virtual **DDS_ReturnCode_t take_next_sample** (**DDS_Octets** &received_data, **DDS_SampleInfo** &sample_info)

*Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).*

^ virtual **DDS_ReturnCode_t return_loan** (**DDS_OctetsSeq** &received_data, **DDS_SampleInfoSeq** &info_seq)

*Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of* `received_data` *and* `info_seq` *obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).*

Static Public Member Functions

^ static **DDSOctetsDataReader** * **narrow** (**DDSDataReader** *reader)

*Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSOctetsDataReader** (p. 1326) pointer.*

6.201.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataReader` < **DDS_Octets** (p. 799) >.

When reading or taking data with this reader, if you request a copy of the samples instead of a loan, and the byte array in a destination data sample is NULL, the middleware will allocate a new array for you of sufficient length to hold the received data. The new array will be allocated with **DDS_OctetBuffer_alloc** (p. 453); the sample's destructor will delete it.

A non- NULL array is assumed to be allocated to sufficient length to store the incoming data. It will not be reallocated.

See also:

FooDataReader (p. 1444)
DDSDataReader (p. 1087)

6.201.2 Member Function Documentation

6.201.2.1 virtual DDS_ReturnCode_t DDSOctetsDataReader::read (DDS_OctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read (p. 1447)

6.201.2.2 virtual DDS_ReturnCode_t DDSOctetsDataReader::take (DDS_OctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data-samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take (p. 1448)

6.201.2.3 virtual DDS_ReturnCode_t DDSOctetsDataReader::read_w_condition (DDS_OctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Accesses via **DDSOctetsDataReader::read** (p. 1328) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_w_condition (p. 1454)

6.201.2.4 virtual DDS_ReturnCode_t DDSOctetsDataReader::take_w_condition (DDS_OctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Analogous to **DDSOctetsDataReader::read_w_condition** (p. 1328) except it accesses samples via the **DDSOctetsDataReader::take** (p. 1328) operation.

See also:

FooDataReader::take_w_condition (p. 1456)

6.201.2.5 virtual DDS_ReturnCode_t DDSOctetsDataReader::read_next_sample (DDS_Octets & *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_next_sample (p. 1457)

6.201.2.6 virtual DDS_ReturnCode_t DDSOctetsDataReader::take_next_sample (DDS_Octets & *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_next_sample (p. 1458)

6.201.2.7 virtual DDS_ReturnCode_t DDSOctetsDataReader::return_loan (DDS_OctetsSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*) [virtual]

Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of `received_data` and `info_seq` obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).

See also:

FooDataReader::return_loan (p. 1471)

6.201.2.8 static DDSOctetsDataReader* DDSOctetsDataReader::narrow (DDSDataReader * *reader*) [static]

Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSOctetsDataReader** (p. 1326) pointer.

See also:

FooDataReader::narrow (p. 1447)

6.202 DDSOctetsDataWriter Class Reference

<<***interface***>> *(*p. *199)* Instantiates `DataWriter` < **DDS_Octets** (p. 799) >.

Inheritance diagram for DDSOctetsDataWriter::

Public Member Functions

- virtual **DDS_ReturnCode_t write** (const **DDS_Octets** &instance_data, const **DDS_InstanceHandle_t** &handle)

  Modifies the value of a ***DDS_Octets*** *(*p. *799) data instance.*

- virtual **DDS_ReturnCode_t write** (const unsigned char *octets, int length, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Modifies the value of a* ***DDS_Octets*** *(*p. *799) data instance.*

- virtual **DDS_ReturnCode_t write** (const **DDS_OctetSeq** &octets, const **DDS_InstanceHandle_t** &handle)

  <<**eXtension**>> (p. 199) *Modifies the value of a* ***DDS_Octets*** *(*p. *799) data instance.*

- virtual **DDS_ReturnCode_t write_w_timestamp** (const **DDS_Octets** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

  Performs the same function as ***DDSOctetsDataWriter::write*** *(*p. *1333) except that it also provides the value for the* `source_timestamp`.

- virtual **DDS_ReturnCode_t write_w_timestamp** (const unsigned char *octets, int length, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

  <<**eXtension**>> (p. 199) *Performs the same function as* ***DDSOctetsDataWriter::write*** *(*p. *1333) except that it also provides the value for the* `source_timestamp`.

- virtual **DDS_ReturnCode_t write_w_timestamp** (const **DDS_OctetSeq** &octets, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

  <<**eXtension**>> (p. 199) *Performs the same function as* ***DDSOctetsDataWriter::write*** *(*p. *1333) except that it also provides the value for the* `source_timestamp`.

- virtual **DDS_ReturnCode_t write_w_params** (const **DDS_Octets** &instance_data, **DDS_WriteParams_t** &params)

  *Performs the same function as **DDSOctetsDataWriter::write*** (p. *1333*) *except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

- virtual **DDS_ReturnCode_t write_w_params** (const unsigned char *octets, int length, **DDS_WriteParams_t** &params)

  <<**eXtension**>> (p. 199) *Performs the same function as **DDSOctetsDataWriter::write*** (p. *1333*) *except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

- virtual **DDS_ReturnCode_t write_w_params** (const **DDS_OctetSeq** &octets, **DDS_WriteParams_t** &params)

  <<**eXtension**>> (p. 199) *Performs the same function as **DDSOctetsDataWriter::write*** (p. *1333*) *except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

Static Public Member Functions

- static **DDSOctetsDataWriter** * **narrow** (**DDSDataWriter** *writer)

  *Narrow the given **DDSDataWriter*** (p. *1113*) *pointer to a **DDSOctetsDataWriter*** (p. *1331*) *pointer.*

6.202.1 Detailed Description

<<***interface***>> (p. *199*) Instantiates `DataWriter` < **DDS_Octets** (p. 799) >.

See also:

FooDataWriter (p. 1475)
DDSDataWriter (p. 1113)

6.202.2 Member Function Documentation

6.202.2.1 static DDSOctetsDataWriter* DDSOctetsDataWriter::narrow (DDSDataWriter * *writer*) [static]

Narrow the given **DDSDataWriter** (p. 1113) pointer to a **DDSOctetsDataWriter** (p. 1331) pointer.

See also:

FooDataWriter::narrow (p. 1477)

6.202.2.2 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write (const DDS_Octets & *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Modifies the value of a **DDS_Octets** (p. 799) data instance.

See also:

FooDataWriter::write (p. 1484)

6.202.2.3 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write (const unsigned char * *octets*, int *length*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<eXtension>> *(p. 199)* Modifies the value of a **DDS_Octets** (p. 799) data instance.

Parameters:

octets *<<in>>* *(p. 200)* Array of octets to be published.

length *<<in>>* *(p. 200)* Number of octets to be published.

handle *<<in>>* *(p. 200)* The special value **DDS_HANDLE_NIL** (p. 55) should be used always.

See also:

FooDataWriter::write (p. 1484)

6.202.2.4 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write (const DDS_OctetSeq & *octets*, const DDS_InstanceHandle_t & *handle*) [virtual]

<<eXtension>> *(p. 199)* Modifies the value of a **DDS_Octets** (p. 799) data instance.

Parameters:

octets *<<in>>* *(p. 200)* Sequence of octets to be published.

handle *<<in>>* *(p. 200)* The special value **DDS_HANDLE_NIL** (p. 55) should be used always.

See also:

FooDataWriter::write (p. 1484)

6.202.2.5 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write_w_timestamp (const DDS_Octets & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSOctetsDataWriter::write** (p. 1333) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.202.2.6 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write_w_timestamp (const unsigned char * *octets*, int *length*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

<<eXtension>> *(p. 199)* Performs the same function as **DDSOctetsDataWriter::write** (p. 1333) except that it also provides the value for the `source_timestamp`.

Parameters:

octets *<<in>>* *(p. 200)* Array of octets to be published.

length *<<in>>* *(p. 200)* Number of octets to be published.

handle *<<in>>* *(p. 200)* The special value **DDS_HANDLE_NIL** (p. 55) should be used always.

source_timestamp *<<in>>* *(p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation. See **FooDataWriter::write_w_timestamp** (p. 1486).

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.202.2.7 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write_w_timestamp (const DDS_OctetSeq & *octets*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

<<eXtension>> *(p. 199)* Performs the same function as **DDSOctetsDataWriter::write** (p. 1333) except that it also provides the value for the `source_timestamp`.

Parameters:

octets *<<in>>* *(p. 200)* Sequence of octets to be published.

handle *<<in>>* *(p. 200)* The special value **DDS_HANDLE_NIL** (p. 55) should be used always.

source_timestamp *<<in>>* *(p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation. See **FooDataWriter::write_w_timestamp** (p. 1486).

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.202.2.8 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write_w_params (const DDS_Octets & *instance_data*, DDS_WriteParams_t & *params*) [virtual]

Performs the same function as **DDSOctetsDataWriter::write** (p. 1333) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

See also:

FooDataWriter::write_w_params (p. 1487)

6.202.2.9 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write_w_params (const unsigned char * *octets*, int *length*, DDS_WriteParams_t & *params*) [virtual]

<<eXtension>> *(p. 199)* Performs the same function as **DDSOctetsDataWriter::write** (p. 1333) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

Parameters:

octets *<<in>> (p. 200)* Array of octets to be published.

length *<<in>> (p. 200)* Number of octets to be published.

params *<<in>> (p. 200)* The **DDS_WriteParams_t** (p. 1067) parameter containing the instance handle, source timestamp, publication priority, and cookie to be used in write operation. See **FooDataWriter::write_w_params** (p. 1487).

See also:

FooDataWriter::write_w_params (p. 1487)

6.202.2.10 virtual DDS_ReturnCode_t DDSOctetsDataWriter::write_w_params (const DDS_OctetSeq & *octets*, DDS_WriteParams_t & *params*) [virtual]

<<eXtension>> (p. 199) Performs the same function as **DDSOctetsDataWriter::write** (p. 1333) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

Parameters:

octets *<<in>> (p. 200)* Sequence of octets to be published.

params *<<in>> (p. 200)* The **DDS_WriteParams_t** (p. 1067) parameter containing the instance handle, source timestamp, publication priority, and cookie to be used in write operation. See **FooDataWriter::write_w_params** (p. 1487).

See also:

FooDataWriter::write_w_params (p. 1487)

6.203 DDSOctetsTypeSupport Class Reference

<<*interface*>> *(p. 199)* **DDS_Octets** (p. 799) type support.

Inheritance diagram for DDSOctetsTypeSupport::

Static Public Member Functions

- static **DDS_ReturnCode_t register_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::Octets")

  *Allows an application to communicate to RTI Connext the existence of the **DDS_Octets** (p. 799) data type.*

- static **DDS_ReturnCode_t unregister_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::Octets")

  *Allows an application to unregister the **DDS_Octets** (p. 799) data type from RTI Connext. After calling unregister_type, no further communication using this type is possible.*

- static const char * **get_type_name** ()

  *Get the default name for the **DDS_Octets** (p. 799) type.*

- static void **print_data** (const **DDS_Octets** *a_data)

  <<**eXtension**>> (p. 199) *Print value of data type to standard out.*

6.203.1 Detailed Description

<<*interface*>> *(p. 199)* **DDS_Octets** (p. 799) type support.

6.203.2 Member Function Documentation

6.203.2.1 static DDS_ReturnCode_t DDSOctetsTypeSupport::register_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::Octets") [static]

Allows an application to communicate to RTI Connext the existence of the **DDS_Octets** (p. 799) data type.

By default, The **DDS_Octets** (p. 799) built-in type is automatically registered when a DomainParticipant is created using the type_name returned by **DDSOctetsTypeSupport::get_type_name** (p. 1339). Therefore, the usage

of this function is optional and it is only required when the automatic built-in type registration is disabled using the participant property "dds.builtin_type.auto_register".

This method can also be used to register the same **DDSOctetsTypeSupport** (p. 1337) with a **DDSDomainParticipant** (p. 1139) using different values for the type_name.

If `register_type` is called multiple times with the same **DDSDomainParticipant** (p. 1139) and `type_name`, the second (and subsequent) registrations are ignored by the operation.

Parameters:

participant *<<in>>* *(p. 200)* the **DDSDomainParticipant** (p. 1139) to register the data type **DDS_Octets** (p. 799) with. Cannot be NULL.

type_name *<<in>>* *(p. 200)* the type name under with the data type **DDS_Octets** (p. 799) is registered with the participant; this type name is used when creating a new **DDSTopic** (p. 1419). (See **DDSDomainParticipant::create_topic** (p. 1175).) The name may not be NULL or longer than 255 characters.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to register a type. Subsequent calls are thread safe.

See also:

DDSDomainParticipant::create_topic (p. 1175)

6.203.2.2 static DDS_ReturnCode_t DDSOctetsTypeSupport::unregister_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::Octets") [static]

Allows an application to unregister the **DDS_Octets** (p. 799) data type from RTI Connext. After calling unregister_type, no further communication using this type is possible.

Precondition:

The **DDS_Octets** (p. 799) type with `type_name` is registered with the participant and all **DDSTopic** (p. 1419) objects referencing the type have been destroyed. If the type is not registered with the participant, or if any **DDSTopic** (p. 1419) is associated with the type, the operation will fail with **DDS_RETCODE_ERROR** (p. 315).

Postcondition:

All information about the type is removed from RTI Connext. No further communication using this type is possible.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to unregister the data type **DDS_Octets** (p. 799) from. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type **DDS_Octets** (p. 799) is registered with the participant. The name should match a name that has been previously used to register a type with the participant. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_BAD_PARAMETER** (p. 315) or **DDS_RETCODE_ERROR** (p. 315)

MT Safety:

SAFE.

See also:

DDSOctetsTypeSupport::register_type (p. 1337)

6.203.2.3 static const char* DDSOctetsTypeSupport::get_type_name () [static]

Get the default name for the **DDS_Octets** (p. 799) type.

Can be used for calling **DDSOctetsTypeSupport::register_type** (p. 1337) or creating **DDSTopic** (p. 1419).

Returns:

default name for the **DDS_Octets** (p. 799) type.

See also:

DDSOctetsTypeSupport::register_type (p. 1337)
DDSDomainParticipant::create_topic (p. 1175)

6.203.2.4 static void DDSOctetsTypeSupport::print_data (const DDS_Octets * *a_data*) [static]

<<***eXtension***>> *(p. 199)* Print value of data type to standard out.

The *generated* implementation of the operation knows how to print value of a data type.

Parameters:

a_data <<***in***>> *(p. 200)* **DDS_Octets** (p. 799) to be printed.

6.204 DDSParticipantBuiltinTopicDataDataReader Class Reference

Instantiates `DataReader` < **DDS_ParticipantBuiltinTopicData** (p. 816) > .

Inheritance diagram for DDSParticipantBuiltinTopicDataDataReader::

6.204.1 Detailed Description

Instantiates `DataReader` < **DDS_ParticipantBuiltinTopicData** (p. 816) > .

DDSDataReader (p. 1087) of topic **DDS_PARTICIPANT_TOPIC_NAME** (p. 285) used for accessing **DDS_ParticipantBuiltinTopicData** (p. 816) of the remote **DDSDomainParticipant** (p. 1139).

Instantiates:

<<*generic*>> *(p. 199)* **FooDataReader** (p. 1444)

See also:

DDS_ParticipantBuiltinTopicData (p. 816)
DDS_PARTICIPANT_TOPIC_NAME (p. 285)

6.205 DDSParticipantBuiltinTopicDataTypeSupport Class Reference

Instantiates `TypeSupport` < **DDS_ParticipantBuiltinTopicData** (p. 816) > .

6.205.1 Detailed Description

Instantiates `TypeSupport` < **DDS_ParticipantBuiltinTopicData** (p. 816) > .

Instantiates:

<<*generic*>> *(p. 199)* **FooTypeSupport** (p. 1509)

See also:

DDS_ParticipantBuiltinTopicData (p. 816)

6.206 DDSPropertyQosPolicyHelper Class Reference

Policy Helpers which facilitate management of the properties in the input policy.

Static Public Member Functions

- static **DDS_Long** **get_number_of_properties** (**DDS_PropertyQosPolicy** &policy)

  Gets the number of properties in the input policy.

- static **DDS_ReturnCode_t** **assert_property** (**DDS_PropertyQosPolicy** &policy, const char *name, const char *value, **DDS_Boolean** propagate)

  Asserts the property identified by name in the input policy.

- static **DDS_ReturnCode_t** **add_property** (**DDS_PropertyQosPolicy** &policy, const char *name, const char *value, **DDS_Boolean** propagate)

  Adds a new property to the input policy.

- static struct **DDS_Property_t** * **lookup_property** (**DDS_PropertyQosPolicy** &policy, const char *name)

  Searches for a property in the input policy given its name.

- static **DDS_ReturnCode_t** **remove_property** (**DDS_PropertyQosPolicy** &policy, const char *name)

  Removes a property from the input policy.

- static **DDS_ReturnCode_t** **get_properties** (**DDS_PropertyQosPolicy** &policy, struct **DDS_PropertySeq** &properties, const char *name_prefix)

  Retrieves a list of properties whose names match the input prefix.

6.206.1 Detailed Description

Policy Helpers which facilitate management of the properties in the input policy.

6.207 DDSPublicationBuiltinTopicDataDataReader Class Reference

Instantiates `DataReader` < **DDS_PublicationBuiltinTopicData** (p. 839) > .

Inheritance diagram for DDSPublicationBuiltinTopicDataDataReader::

6.207.1 Detailed Description

Instantiates `DataReader` < **DDS_PublicationBuiltinTopicData** (p. 839) > .

DDSDataReader (p. 1087) of topic **DDS_PUBLICATION_TOPIC_NAME** (p. 289) used for accessing **DDS_PublicationBuiltinTopicData** (p. 839) of the remote **DDSDataWriter** (p. 1113) and the associated **DDSPublisher** (p. 1346).

Instantiates:

<<*generic*>> *(p. 199)* **FooDataReader** (p. 1444)

See also:

DDS_PublicationBuiltinTopicData (p. 839)
DDS_PUBLICATION_TOPIC_NAME (p. 289)

6.208 DDSPublicationBuiltinTopicDataTypeSupport Class Reference

Instantiates `TypeSupport` < **DDS_PublicationBuiltinTopicData** (p. 839) > .

6.208.1 Detailed Description

Instantiates `TypeSupport` < **DDS_PublicationBuiltinTopicData** (p. 839) > .

Instantiates:

<<*generic*>> *(p. 199)* **FooTypeSupport** (p. 1509)

See also:

DDS_PublicationBuiltinTopicData (p. 839)

6.209 DDSPublisher Class Reference

<<*interface*>> *(p. 199)* A publisher is the object responsible for the actual dissemination of publications.

Inheritance diagram for DDSPublisher::

Public Member Functions

- virtual **DDS_ReturnCode_t get_default_datawriter_qos** (**DDS_DataWriterQos** &qos)=0

  *Copies the default **DDS_DataWriterQos** (p. 553) values into the provided **DDS_DataWriterQos** (p. 553) instance.*

- virtual **DDS_ReturnCode_t set_default_datawriter_qos** (const **DDS_DataWriterQos** &qos)=0

  *Sets the default **DDS_DataWriterQos** (p. 553) values for this publisher.*

- virtual **DDS_ReturnCode_t set_default_datawriter_qos_with_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Set the default **DDS_DataWriterQos** (p. 553) values for this publisher based on the input XML QoS profile.*

- virtual **DDS_ReturnCode_t set_default_library** (const char *library_name)=0

  <<**eXtension**>> (p. 199) *Sets the default XML library for a **DDSPublisher** (p. 1346).*

- virtual const char * **get_default_library** ()=0

  <<**eXtension**>> (p. 199) *Gets the default XML library associated with a **DDSPublisher** (p. 1346).*

- virtual **DDS_ReturnCode_t set_default_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Sets the default XML profile for a **DDSPublisher** (p. 1346).*

- virtual const char * **get_default_profile** ()=0

  <<**eXtension**>> (p. 199) *Gets the default XML profile associated with a **DDSPublisher** (p. 1346).*

- virtual const char * **get_default_profile_library** ()=0

<<**eXtension**>> (p. 199) *Gets the library where the default XML QoS profile is contained for a* ***DDSPublisher*** *(p. 1346).*

^ virtual **DDSDataWriter** * **create_datawriter** (**DDSTopic** *topic, const **DDS_DataWriterQos** &qos, **DDSDataWriterListener** *listener, **DDS_StatusMask** mask)=0

Creates a ***DDSDataWriter*** *(p. 1113) that will be attached and belong to the* ***DDSPublisher*** *(p. 1346).*

^ virtual **DDSDataWriter** * **create_datawriter_with_profile** (**DDSTopic** *topic, const char *library_name, const char *profile_name, **DDSDataWriterListener** *listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a* ***DDSDataWriter*** *(p. 1113) object using the* ***DDS_DataWriterQos*** *(p. 553) associated with the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t delete_datawriter** (**DDSDataWriter** *a_datawriter)=0

Deletes a ***DDSDataWriter*** *(p. 1113) that belongs to the* ***DDSPublisher*** *(p. 1346).*

^ virtual **DDSDataWriter** * **lookup_datawriter** (const char *topic_name)=0

Retrieves the ***DDSDataWriter*** *(p. 1113) for a specific* ***DDSTopic*** *(p. 1419).*

^ virtual **DDS_ReturnCode_t get_all_datawriters** (DDSDataWriterSeq &writers)=0

Retrieve all the DataWriters created from this Publisher.

^ virtual **DDS_ReturnCode_t suspend_publications** ()=0

Indicates to RTI Connext that the application is about to make multiple modifications using ***DDSDataWriter*** *(p. 1113) objects belonging to the* ***DDSPublisher*** *(p. 1346).*

^ virtual **DDS_ReturnCode_t resume_publications** ()=0

Indicates to RTI Connext that the application has completed the multiple changes initiated by the previous ***DDSPublisher::suspend_publications*** *(p. 1360).*

^ virtual **DDS_ReturnCode_t begin_coherent_changes** ()=0

Indicates that the application will begin a coherent set of modifications using ***DDSDataWriter*** *(p. 1113) objects attached to the* ***DDSPublisher*** *(p. 1346).*

^ virtual **DDS_ReturnCode_t end_coherent_changes** ()=0

*Terminates the coherent set initiated by the matching call to **DDSPublisher::begin_coherent_changes*** (p. 1362).

^ virtual **DDSDomainParticipant** * **get_participant** ()=0

*Returns the **DDSDomainParticipant*** (p. 1139) *to which the **DDSPublisher*** (p. 1346) *belongs.*

^ virtual **DDS_ReturnCode_t delete_contained_entities** ()=0

*Deletes all the entities that were created by means of the "create" operation on the **DDSPublisher*** (p. 1346).

^ virtual **DDS_ReturnCode_t copy_from_topic_qos** (**DDS_DataWriterQos** &a_datawriter_qos, const **DDS_TopicQos** &a_topic_qos)=0

*Copies the policies in the **DDS_TopicQos*** (p. 965) *to the corresponding policies in the **DDS_DataWriterQos*** (p. 553).

^ virtual **DDS_ReturnCode_t wait_for_acknowledgments** (const **DDS_Duration_t** &max_wait)=0

Blocks the calling thread until all data written by the Publisher's reliable DataWriters is acknowledged, or until timeout expires.

^ virtual **DDS_ReturnCode_t wait_for_asynchronous_publishing** (const **DDS_Duration_t** &max_wait)=0

<<**eXtension**>> (p. 199) *Blocks the calling thread until asynchronous sending is complete.*

^ virtual **DDS_ReturnCode_t set_qos** (const **DDS_PublisherQos** &qos)=0

Sets the publisher QoS.

^ virtual **DDS_ReturnCode_t set_qos_with_profile** (const char *library_name, const char *profile_name)=0

<<**eXtension**>> (p. 199) *Change the QoS of this publisher using the input XML QoS profile.*

^ virtual **DDS_ReturnCode_t get_qos** (**DDS_PublisherQos** &qos)=0

Gets the publisher QoS.

^ virtual **DDS_ReturnCode_t set_listener** (**DDSPublisherListener** *l, **DDS_StatusMask** mask=DDS_STATUS_MASK_ALL)=0

Sets the publisher listener.

- virtual **DDSPublisherListener** * **get_listener** ()=0

  Get the publisher listener.

- virtual **DDSDataWriter** * **lookup_datawriter_by_name_exp** (const char *datawriter_name)=0

  <<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Retrieves a **DDSDataWriter** (p. 1113) contained within the **DDSPublisher** (p. 1346) the **DDSDataWriter** (p. 1113) entity name.*

6.209.1 Detailed Description

<<***interface***>> *(p. 199)* A publisher is the object responsible for the actual dissemination of publications.

QoS:

DDS_PublisherQos (p. 851)

Listener:

DDSPublisherListener (p. 1370)

A publisher acts on the behalf of one or several **DDSDataWriter** (p. 1113) objects that belong to it. When it is informed of a change to the data associated with one of its **DDSDataWriter** (p. 1113) objects, it decides when it is appropriate to actually send the data-update message. In making this decision, it considers any extra information that goes with the data (timestamp, writer, etc.) as well as the QoS of the **DDSPublisher** (p. 1346) and the **DDSDataWriter** (p. 1113).

The following operations may be called even if the **DDSPublisher** (p. 1346) is not enabled. Other operations will fail with the value **DDS_RETCODE_NOT_ENABLED** (p. 315) if called on a disabled **DDSPublisher** (p. 1346):

- The base-class operations **DDSPublisher::set_qos** (p. 1366), **DDSPublisher::set_qos_with_profile** (p. 1366), **DDSPublisher::get_qos** (p. 1367), **DDSPublisher::set_listener** (p. 1368), **DDSPublisher::get_listener** (p. 1368), **DDSEntity::enable** (p. 1256), **DDSEntity::get_statuscondition** (p. 1257), **DDSEntity::get_status_changes** (p. 1257)
- **DDSPublisher::create_datawriter** (p. 1355), **DDSPublisher::create_datawriter_with_profile** (p. 1357), **DDSPublisher::delete_datawriter** (p. 1358), **DDSPublisher::delete_contained_entities** (p. 1363), **DDSPublisher::set_default_**

datawriter_qos (p. 1351), **DDSPublisher::set_default_datawriter_qos_with_profile** (p. 1351), **DDSPublisher::get_default_datawriter_qos** (p. 1350), **DDSPublisher::wait_for_acknowledgments** (p. 1364), **DDSPublisher::set_default_library** (p. 1352), **DDSPublisher::set_default_profile** (p. 1353),

See also:

Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_publisher.cxx.

6.209.2 Member Function Documentation

6.209.2.1 virtual DDS_ReturnCode_t DDSPublisher::get_default_datawriter_qos (DDS_DataWriterQos & *qos*) [pure virtual]

Copies the default **DDS_DataWriterQos** (p. 553) values into the provided **DDS_DataWriterQos** (p. 553) instance.

The retreived `qos` will match the set of values specified on the last successful call to **DDSPublisher::set_default_datawriter_qos** (p. 1351) or **DDSPublisher::set_default_datawriter_qos_with_profile** (p. 1351), or else, if the call was never made, the default values from is owning **DDSDomainParticipant** (p. 1139).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

MT Safety:

UNSAFE. It is not safe to retrieve the default QoS value from a **DDSPublisher** (p. 1346) while another thread may be simultaneously calling **DDSPublisher::set_default_datawriter_qos** (p. 1351).

Parameters:

qos *<<inout>> (p. 200)* **DDS_DataWriterQos** (p. 553) to be filled-up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DATAWRITER_QOS_DEFAULT (p. 84)
DDSPublisher::create_datawriter (p. 1355)

6.209.2.2 virtual DDS_ReturnCode_t DDSPublisher::set_default_datawriter_qos (const DDS_DataWriterQos & *qos*) [pure virtual]

Sets the default **DDS_DataWriterQos** (p. 553) values for this publisher.

This call causes the default values inherited from the owning **DDSDomainParticipant** (p. 1139) to be overridden.

This default value will be used for newly created **DDSDataWriter** (p. 1113) if **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) is specified as the `qos` parameter when **DDSPublisher::create_datawriter** (p. 1355) is called.

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default QoS value from a **DDSPublisher** (p. 1346) while another thread may be simultaneously calling **DDSPublisher::set_default_datawriter_qos** (p. 1351), **DDSPublisher::get_default_datawriter_qos** (p. 1350) or calling **DDSPublisher::create_datawriter** (p. 1355) with **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) as the `qos` parameter.

Parameters:

qos <<*in*>> *(p. 200)* Default qos to be set. The special value **DDS_DATAREADER_QOS_DEFAULT** (p. 99) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSPublisher::set_default_datawriter_qos** (p. 1351) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.209.2.3 virtual DDS_ReturnCode_t DDSPublisher::set_default_datawriter_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<*eXtension*>> *(p. 199)* Set the default **DDS_DataWriterQos** (p. 553) values for this publisher based on the input XML QoS profile.

This default value will be used for newly created **DDSDataWriter** (p. 1113) if **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) is specified as the `qos` parameter when **DDSPublisher::create_datawriter** (p. 1355) is called.

Precondition:

The **DDS_DataWriterQos** (p. 553) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default QoS value from a **DDSPublisher** (p. 1346) while another thread may be simultaneously calling **DDSPublisher::set_default_datawriter_qos** (p. 1351), **DDSPublisher::get_default_datawriter_qos** (p. 1350) or calling **DDSPublisher::create_datawriter** (p. 1355) with **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) as the `qos` parameter.

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSPublisher::set_default_library** (p. 1352)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSPublisher::set_default_profile** (p. 1353)).

If the input profile cannot be found, the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_DATAWRITER_QOS_DEFAULT (p. 84)
DDSPublisher::create_datawriter_with_profile (p. 1357)

6.209.2.4 virtual DDS_ReturnCode_t DDSPublisher::set_default_library (const char * *library_name*) [pure virtual]

<<eXtension>> (p. 199) Sets the default XML library for a **DDSPublisher** (p. 1346).

This method specifies the library that will be used as the default the next time a default library is needed during a call to one of this Publisher's operations.

Any API requiring a library_name as a parameter can use null to refer to the default library.

If the default library is not set, the **DDSPublisher** (p. 1346) inherits the default from the **DDSDomainParticipant** (p. 1139) (see **DDSDomainParticipant::set_default_library** (p. 1159)).

Parameters:

library_name *<<in>>* *(p. 200)* Library name. If library_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSPublisher::get_default_library (p. 1353)

6.209.2.5 virtual const char* DDSPublisher::get_default_library () [pure virtual]

<<eXtension>> *(p. 199)* Gets the default XML library associated with a **DDSPublisher** (p. 1346).

Returns:

The default library or null if the default library was not set.

See also:

DDSPublisher::set_default_library (p. 1352)

6.209.2.6 virtual DDS_ReturnCode_t DDSPublisher::set_default_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Sets the default XML profile for a **DDSPublisher** (p. 1346).

This method specifies the profile that will be used as the default the next time a default Publisher profile is needed during a call to one of this Publisher's

operations. When calling a **DDSPublisher** (p. 1346) method that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

If the default profile is not set, the **DDSPublisher** (p. 1346) inherits the default from the **DDSDomainParticipant** (p. 1139) (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

This method does not set the default QoS for **DDSDataWriter** (p. 1113) objects created by the **DDSPublisher** (p. 1346); for this functionality, use **DDSPublisher::set_default_datawriter_qos_with_profile** (p. 1351) (you may pass in NULL after having called **set_default_profile()** (p. 1353)).

This method does not set the default QoS for newly created Publishers; for this functionality, use **DDSDomainParticipant::set_default_publisher_qos_with_profile** (p. 1165).

Parameters:

library_name *<<in>> (p. 200)* The library name containing the profile.

profile_name *<<in>> (p. 200)* The profile name. If profile_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSPublisher::get_default_profile (p. 1354)
DDSPublisher::get_default_profile_library (p. 1355)

6.209.2.7 virtual const char* DDSPublisher::get_default_profile () [pure virtual]

<<eXtension>> (p. 199) Gets the default XML profile associated with a **DDSPublisher** (p. 1346).

Returns:

The default profile or null if the default profile was not set.

See also:

DDSPublisher::set_default_profile (p. 1353)

6.209.2.8 virtual const char* DDSPublisher::get_default_profile_library () [pure virtual]

<<*eXtension*>> *(p. 199)* Gets the library where the default XML QoS profile is contained for a **DDSPublisher** (p. 1346).

The default profile library is automatically set when **DDSPublisher::set_default_profile** (p. 1353) is called.

This library can be different than the **DDSPublisher** (p. 1346) default library (see **DDSPublisher::get_default_library** (p. 1353)).

Returns:

The default profile library or null if the default profile was not set.

See also:

DDSPublisher::set_default_profile (p. 1353)

6.209.2.9 virtual DDSDataWriter* DDSPublisher::create_datawriter (DDSTopic * *topic*, const DDS_DataWriterQos & *qos*, DDSDataWriterListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

Creates a **DDSDataWriter** (p. 1113) that will be attached and belong to the **DDSPublisher** (p. 1346).

For each application-defined type, `Foo` (p. 1443), there is an implied, auto-generated class `FooDataWriter` (p. 1475) that extends **DDSDataWriter** (p. 1113) and contains the operations to write data of type `Foo` (p. 1443).

Note that a common application pattern to construct the QoS for the **DDSDataWriter** (p. 1113) is to:

- Retrieve the QoS policies on the associated **DDSTopic** (p. 1419) by means of the **DDSTopic::get_qos** (p. 1423) operation.
- Retrieve the default **DDSDataWriter** (p. 1113) qos by means of the **DDSPublisher::get_default_datawriter_qos** (p. 1350) operation.
- Combine those two QoS policies (for example, using **DDSPublisher::copy_from_topic_qos** (p. 1364)) and selectively modify policies as desired.

When a **DDSDataWriter** (p. 1113) is created, only those transports already registered are available to the **DDSDataWriter** (p. 1113). See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

Precondition:

If publisher is enabled, topic must have been enabled. Otherwise, this operation will fail and no **DDSDataWriter** (p. 1113) will be created.
The given **DDSTopic** (p. 1419) must have been created from the same participant as this publisher. If it was created from a different participant, this method will fail.

MT Safety:

UNSAFE. If **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) is used for the `qos` parameter, it is not safe to create the datawriter while another thread may be simultaneously calling **DDSPublisher::set_default_datawriter_qos** (p. 1351).

Parameters:

topic *<<in>>* *(p. 200)* The **DDSTopic** (p. 1419) that the **DDSDataWriter** (p. 1113) will be associated with. Cannot be NULL.

qos *<<in>>* *(p. 200)* QoS to be used for creating the new **DDSDataWriter** (p. 1113). The special value **DDS_DATAWRITER_QOS_DEFAULT** (p. 84) can be used to indicate that the **DDSDataWriter** (p. 1113) should be created with the default **DDS_DataWriterQos** (p. 553) set in the **DDSPublisher** (p. 1346). The special value **DDS_DATAWRITER_QOS_USE_TOPIC_QOS** (p. 84) can be used to indicate that the **DDSDataWriter** (p. 1113) should be created with the combination of the default **DDS_DataWriterQos** (p. 553) set on the **DDSPublisher** (p. 1346) and the **DDS_TopicQos** (p. 965) of the **DDSTopic** (p. 1419).

listener *<<in>>* *(p. 200)* The listener of the **DDSDataWriter** (p. 1113).

mask *<<in>>* *(p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataWriter** (p. 1113) of a derived class specific to the data type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataWriter (p. 1475)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataWriterQos (p. 553) for rules on consistency among QoS
DDS_DATAWRITER_QOS_DEFAULT (p. 84)
DDS_DATAWRITER_QOS_USE_TOPIC_QOS (p. 84)

DDSPublisher::create_datawriter_with_profile (p. 1357)
DDSPublisher::get_default_datawriter_qos (p. 1350)
DDSTopic::set_qos (p. 1421)
DDSPublisher::copy_from_topic_qos (p. 1364)
DDSDataWriter::set_listener (p. 1128)

Examples:

HelloWorld_publisher.cxx.

6.209.2.10 virtual DDSDataWriter* DDSPublisher::create_datawriter_with_profile (DDSTopic * *topic*, const char * *library_name*, const char * *profile_name*, DDSDataWriterListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<eXtension>> (p. 199) Creates a **DDSDataWriter** (p. 1113) object using the **DDS_DataWriterQos** (p. 553) associated with the input XML QoS profile.

The **DDSDataWriter** (p. 1113) will be attached and belong to the **DDSPublisher** (p. 1346).

For each application-defined type, `Foo` (p. 1443), there is an implied, auto-generated class `FooDataWriter` (p. 1475) that extends **DDSDataWriter** (p. 1113) and contains the operations to write data of type `Foo` (p. 1443).

When a **DDSDataWriter** (p. 1113) is created, only those transports already registered are available to the **DDSDataWriter** (p. 1113). See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

Precondition:

If publisher is enabled, topic must have been enabled. Otherwise, this operation will fail and no **DDSDataWriter** (p. 1113) will be created.
The given **DDSTopic** (p. 1419) must have been created from the same participant as this publisher. If it was created from a different participant, this method will return NULL.

Parameters:

topic *<<in>> (p. 200)* The **DDSTopic** (p. 1419) that the **DDSDataWriter** (p. 1113) will be associated with. Cannot be NULL.

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSPublisher::set_default_library** (p. 1352)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSPublisher::set_default_profile** (p. 1353)).

listener *<<in>>* *(p. 200)* The listener of the **DDSDataWriter** (p. 1113).

mask *<<in>>* *(p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataWriter** (p. 1113) of a derived class specific to the data type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataWriter (p. 1475)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataWriterQos (p. 553) for rules on consistency among QoS
DDSPublisher::create_datawriter (p. 1355)
DDSPublisher::get_default_datawriter_qos (p. 1350)
DDSTopic::set_qos (p. 1421)
DDSPublisher::copy_from_topic_qos (p. 1364)
DDSDataWriter::set_listener (p. 1128)

6.209.2.11 virtual DDS_ReturnCode_t DDSPublisher::delete_datawriter (DDSDataWriter * *a_datawriter*) [pure virtual]

Deletes a **DDSDataWriter** (p. 1113) that belongs to the **DDSPublisher** (p. 1346).

The deletion of the **DDSDataWriter** (p. 1113) will automatically unregister all instances. Depending on the settings of the **WRITER_DATA_LIFECYCLE** (p. 375) QosPolicy, the deletion of the **DDSDataWriter** (p. 1113) may also dispose all instances.

6.209.3 Special Instructions if Using 'Timestamp' APIs and BY_SOURCE_TIMESTAMP Destination Ordering:

If the DataWriter's **DDS_DestinationOrderQosPolicy::kind** (p. 572) is **DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS** (p. 366), calls to **delete_datawriter()** (p. 1358) may fail if your application

has previously used the 'with timestamp' APIs (write_w_timestamp(), register_instance_w_timestamp(), unregister_instance_w_timestamp(), or dispose_w_timestamp()) with a timestamp larger (later) than the time at which **delete_datawriter()** (p. 1358) is called. To prevent **delete_datawriter()** (p. 1358) from failing in this situation, either:

- Change the **WRITER_DATA_LIFECYCLE** (p. 375) QosPolicy so that RTI Connext will not autodispose unregistered instances (set **DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances** (p. 1072) to **DDS_BOOLEAN_FALSE** (p. 299).) or
- Explicitly call unregister_instance_w_timestamp() for all instances modified with the *_w_timestamp() APIs before calling **delete_datawriter()** (p. 1358).

Precondition:

If the **DDSDataWriter** (p. 1113) does not belong to the **DDSPublisher** (p. 1346), the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSDataWriter** (p. 1113) will not be called after this method completes successfully.

Parameters:

a_datawriter *<<in>> (p. 200)* The **DDSDataWriter** (p. 1113) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.209.3.1 virtual DDSDataWriter* DDSPublisher::lookup_datawriter (const char * *topic_name*) [pure virtual]

Retrieves the **DDSDataWriter** (p. 1113) for a specific **DDSTopic** (p. 1419).

This returned **DDSDataWriter** (p. 1113) is either enabled or disabled.

Parameters:

topic_name *<<in>> (p. 200)* Name of the **DDSTopic** (p. 1419) associated with the **DDSDataWriter** (p. 1113) that is to be looked up. Cannot be NULL.

Returns:

A **DDSDataWriter** (p. 1113) that belongs to the **DDSPublisher** (p. 1346) attached to the **DDSTopic** (p. 1419) with `topic_name`. If no such **DDSDataWriter** (p. 1113) exists, this operation returns NULL.

If more than one **DDSDataWriter** (p. 1113) is attached to the **DDSPublisher** (p. 1346) with the same `topic_name`, then this operation may return any one of them.

MT Safety:

UNSAFE. It is not safe to lookup a **DDSDataWriter** (p. 1113) in one thread while another thread is simultaneously creating or destroying that **DDSDataWriter** (p. 1113).

6.209.3.2 virtual DDS_ReturnCode_t DDSPublisher::get_all_datawriters (DDSDataWriterSeq & *writers*) [pure virtual]

Retrieve all the DataWriters created from this Publisher.

Parameters:

writers <<***inout***>> *(p. 200)* Sequence where the DataWriters will be added

Returns:

One of the **Standard Return Codes** (p. 314)

6.209.3.3 virtual DDS_ReturnCode_t DDSPublisher::suspend_publications () [pure virtual]

Indicates to RTI Connext that the application is about to make multiple modifications using **DDSDataWriter** (p. 1113) objects belonging to the **DDSPublisher** (p. 1346).

It is a **hint** to RTI Connext so it can optimize its performance by e.g., holding the dissemination of the modifications and then batching them.

The use of this operation must be matched by a corresponding call to **DDSPublisher::resume_publications** (p. 1361) indicating that the set of modifications has completed.

If the **DDSPublisher** (p. 1346) is deleted before **DDSPublisher::resume_publications** (p. 1361) is called, any suspended updates yet to be published will be discarded.

RTI Connext is not required and does not currently make use of this hint in any way. However, similar results can be achieved by using *asynchronous publishing*. Combined with **DDSFlowController** (p. 1259), **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) **DDSDataWriter** (p. 1113) instances allow the user even finer control of traffic shaping and sample coalescing.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

DDSFlowController (p. 1259)
DDSFlowController::trigger_flow (p. 1261)
DDS_ON_DEMAND_FLOW_CONTROLLER_NAME (p. 93)
DDS_PublishModeQosPolicy (p. 853)

6.209.3.4 virtual DDS_ReturnCode_t DDSPublisher::resume_publications () [pure virtual]

Indicates to RTI Connext that the application has completed the multiple changes initiated by the previous **DDSPublisher::suspend_publications** (p. 1360).

This is a **hint** to RTI Connext that can be used for example, to batch all the modifications made since the **DDSPublisher::suspend_publications** (p. 1360).

RTI Connext is not required and does not currently make use of this hint in any way. However, similar results can be achieved by using *asynchronous publishing*. Combined with **DDSFlowController** (p. 1259), **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422) **DDSDataWriter** (p. 1113) instances allow the user even finer control of traffic shaping and sample coalescing.

Precondition:

A call to **DDSPublisher::resume_publications** (p. 1361) must match a previous call to **DDSPublisher::suspend_publications** (p. 1360). Otherwise the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

DDSFlowController (p. 1259)
DDSFlowController::trigger_flow (p. 1261)
DDS_ON_DEMAND_FLOW_CONTROLLER_NAME (p. 93)
DDS_PublishModeQosPolicy (p. 853)

6.209.3.5 virtual DDS_ReturnCode_t DDSPublisher::begin_coherent_changes () [pure virtual]

Indicates that the application will begin a coherent set of modifications using **DDSDataWriter** (p. 1113) objects attached to the **DDSPublisher** (p. 1346).

A 'coherent set' is a set of modifications that must be propagated in such a way that they are interpreted at the receiver's side as a consistent set of modifications; that is, the receiver will only be able to access the data after all the modifications in the set are available at the receiver end.

A connectivity change may occur in the middle of a set of coherent changes; for example, the set of partitions used by the **DDSPublisher** (p. 1346) or one of its **DDSSubscriber** (p. 1390) s may change, a late-joining **DDSDataReader** (p. 1087) may appear on the network, or a communication failure may occur. In the event that such a change prevents an entity from receiving the entire set of coherent changes, that entity must behave as if it had received none of the set.

These calls can be nested. In that case, the coherent set terminates only with the last call to **DDSPublisher::end_coherent_changes** (p. 1362).

The support for coherent changes enables a publishing application to change the value of several data-instances that could belong to the same or different topics and have those changes be seen *atomically* by the readers. This is useful in cases where the values are inter-related (for example, if there are two data-instances representing the altitude and velocity vector of the same aircraft and both are changed, it may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.209.3.6 virtual DDS_ReturnCode_t DDSPublisher::end_coherent_changes () [pure virtual]

Terminates the coherent set initiated by the matching call to **DDSPublisher::begin_coherent_changes** (p. 1362).

Precondition:

If there is no matching call to **DDSPublisher::begin_coherent_changes** (p. 1362) the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.209.3.7 virtual DDSDomainParticipant* DDSPublisher::get_participant () [pure virtual]

Returns the **DDSDomainParticipant** (p. 1139) to which the **DDSPublisher** (p. 1346) belongs.

Returns:

the **DDSDomainParticipant** (p. 1139) to which the **DDSPublisher** (p. 1346) belongs.

6.209.3.8 virtual DDS_ReturnCode_t DDSPublisher::delete_contained_entities () [pure virtual]

Deletes all the entities that were created by means of the "create" operation on the **DDSPublisher** (p. 1346).

Deletes all contained **DDSDataWriter** (p. 1113) objects. Once **DDSPublisher::delete_contained_entities** (p. 1363) completes successfully, the application may delete the **DDSPublisher** (p. 1346), knowing that it has no contained **DDSDataWriter** (p. 1113) objects.

The operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) if any of the contained entities is in a state where it cannot be deleted.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.209.3.9 virtual DDS_ReturnCode_t DDSPublisher::copy_from_topic_qos (DDS_DataWriterQos & *a_datawriter_qos*, const DDS_TopicQos & *a_topic_qos*) [pure virtual]

Copies the policies in the **DDS_TopicQos** (p. 965) to the corresponding policies in the **DDS_DataWriterQos** (p. 553).

Copies the policies in the **DDS_TopicQos** (p. 965) to the corresponding policies in the **DDS_DataWriterQos** (p. 553) (replacing values in the **DDS_DataWriterQos** (p. 553), if present).

This is a "convenience" operation most useful in combination with the operations **DDSPublisher::get_default_datawriter_qos** (p. 1350) and **DDSTopic::get_qos** (p. 1423). The operation **DDSPublisher::copy_from_topic_qos** (p. 1364) can be used to merge the **DDSDataWriter** (p. 1113) default QoS policies with the corresponding ones on the **DDSTopic** (p. 1419). The resulting QoS can then be used to create a new **DDSDataWriter** (p. 1113), or set its QoS.

This operation does not check the resulting **DDS_DataWriterQos** (p. 553) for consistency. This is because the 'merged' **DDS_DataWriterQos** (p. 553) may not be the final one, as the application can still modify some policies prior to applying the policies to the **DDSDataWriter** (p. 1113).

Parameters:

a_datawriter_qos *<<inout>> (p. 200)* **DDS_DataWriterQos** (p. 553) to be filled-up.

a_topic_qos *<<in>> (p. 200)* **DDS_TopicQos** (p. 965) to be merged with **DDS_DataWriterQos** (p. 553).

Returns:

One of the **Standard Return Codes** (p. 314)

6.209.3.10 virtual DDS_ReturnCode_t DDSPublisher::wait_for_acknowledgments (const DDS_Duration_t & *max_wait*) [pure virtual]

Blocks the calling thread until all data written by the Publisher's reliable DataWriters is acknowledged, or until timeout expires.

This operation blocks the calling thread until either all data written by the reliable entities is acknowledged by (a) all reliable **DDSDataReader** (p. 1087) entities that are matched and alive and (b) by all required subscriptions, or until the duration specified by the `max_wait` parameter elapses, whichever happens first. A successful completion indicates that all the samples written have been

acknowledged; a return value of TIMEOUT indicates that max_wait elapsed before all the data was acknowledged.

Note that if a thread is blocked in the call to this operation on a **DDSPublisher** (p. 1346) and a different thread writes new samples on any of the reliable DataWriters that belong to this Publisher, the new samples must be acknowledged before unblocking the thread that is waiting on this operation.

If none of the **DDSDataWriter** (p. 1113) instances have **DDS_ReliabilityQosPolicy** (p. 865) kind set to RELIABLE, the operation will complete successfully.

Parameters:

max_wait *<<in>> (p. 200)* Specifies maximum time to wait for acknowledgements **DDS_Duration_t** (p. 621) .

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_NOT_ENABLED** (p. 315), **DDS_RETCODE_TIMEOUT** (p. 316)

6.209.3.11 virtual DDS_ReturnCode_t DDSPublisher::wait_for_asynchronous_publishing (const DDS_Duration_t & *max_wait*) [pure virtual]

<<eXtension>> (p. 199) Blocks the calling thread until asynchronous sending is complete.

This operation blocks the calling thread (up to `max_wait`) until all data written by the asynchronous **DDSDataWriter** (p. 1113) entities is sent and acknowledged (if reliable) by all matched **DDSDataReader** (p. 1087) entities. A successful completion indicates that all the samples written have been sent and acknowledged where applicable; if it times out, this indicates that `max_wait` elapsed before all the data was sent and/or acknowledged.

In other words, this guarantees that sending to best effort **DDSDataReader** (p. 1087) is complete in addition to what **DDSPublisher::wait_for_acknowledgments** (p. 1364) provides.

If none of the **DDSDataWriter** (p. 1113) instances have **DDS_PublishModeQosPolicy::kind** (p. 855) set to **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422), the operation will complete immediately, with **DDS_RETCODE_OK** (p. 315).

Parameters:

max_wait *<<in>> (p. 200)* Specifies maximum time to wait for acknowledgements **DDS_Duration_t** (p. 621).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_NOT_ENABLED** (p. 315), **DDS_RETCODE_TIMEOUT** (p. 316)

6.209.3.12 virtual DDS_ReturnCode_t DDSPublisher::set_qos (const DDS_PublisherQos & *qos*) [pure virtual]

Sets the publisher QoS.

This operation modifies the QoS of the **DDSPublisher** (p. 1346).

The **DDS_PublisherQos::group_data** (p. 852), **DDS_PublisherQos::partition** (p. 852) and **DDS_PublisherQos::entity_factory** (p. 852) can be changed. The other policies are immutable.

Parameters:

qos *<<**in**>> (p. 200)* **DDS_PublisherQos** (p. 851) to be set to. Policies must be consistent. Immutable policies cannot be changed after **DDSPublisher** (p. 1346) is enabled. The special value **DDS_PUBLISHER_QOS_DEFAULT** (p. 38) can be used to indicate that the QoS of the **DDSPublisher** (p. 1346) should be changed to match the current default **DDS_PublisherQos** (p. 851) set in the **DDSDomainParticipant** (p. 1139).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_PublisherQos (p. 851) for rules on consistency among QoS
set_qos (abstract) (p. 1254)
Operations Allowed in Listener Callbacks (p. 1320)

6.209.3.13 virtual DDS_ReturnCode_t DDSPublisher::set_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

*<<**eXtension**>> (p. 199)* Change the QoS of this publisher using the input XML QoS profile.

This operation modifies the QoS of the **DDSPublisher** (p. 1346).

The **DDS_PublisherQos::group_data** (p. 852), **DDS_PublisherQos::partition** (p. 852) and **DDS_PublisherQos::entity_factory** (p. 852) can be changed. The other policies are immutable.

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSPublisher::set_default_library** (p. 1352)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSPublisher::set_default_profile** (p. 1353)).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_PublisherQos (p. 851) for rules on consistency among QoS
Operations Allowed in Listener Callbacks (p. 1320)

6.209.3.14 virtual DDS_ReturnCode_t DDSPublisher::get_qos (DDS_PublisherQos & *qos*) [pure virtual]

Gets the publisher QoS.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos *<<in>> (p. 200)* **DDS_PublisherQos** (p. 851) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

get_qos (abstract) (p. 1255)

6.209.3.15 virtual DDS_ReturnCode_t DDSPublisher::set_listener (DDSPublisherListener * *l*, DDS_StatusMask *mask* = DDS_STATUS_MASK_ALL) [pure virtual]

Sets the publisher listener.

Parameters:

l <<*in*>> *(p. 200)* **DDSPublisherListener** (p. 1370) to set to.

mask <<*in*>> *(p. 200)* **DDS_StatusMask** (p. 321) associated with the **DDSPublisherListener** (p. 1370).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

set_listener (abstract) (p. 1255)

6.209.3.16 virtual DDSPublisherListener* DDSPublisher::get_listener () [pure virtual]

Get the publisher listener.

Returns:

DDSPublisherListener (p. 1370) of the **DDSPublisher** (p. 1346).

See also:

get_listener (abstract) (p. 1256)

6.209.3.17 virtual DDSDataWriter* DDSPublisher::lookup_datawriter_by_name_exp (const char * *datawriter_name*) [pure virtual]

<<*experimental*>> *(p. 199)* <<*eXtension*>> *(p. 199)* Retrieves a **DDSDataWriter** (p. 1113) contained within the **DDSPublisher** (p. 1346) the **DDSDataWriter** (p. 1113) entity name.

Every **DDSDataWriter** (p. 1113) in the system has an entity name which is configured and stored in the <<*eXtension*>> *(p. 199)* EntityName policy, **ENTITY_NAME** (p. 445).

This operation retrieves the **DDSDataWriter** (p. 1113) within the **DDSPublisher** (p. 1346) whose name matches the one specified. If there are several **DDSDataWriter** (p. 1113) with the same name within the **DDSPublisher** (p. 1346), the operation returns the first matching occurrence.

Parameters:

datawriter_name *<<in>>* *(p. 200)* Entity name of the **DDSDataWriter** (p. 1113).

Returns:

The first **DDSDataWriter** (p. 1113) found with the specified name or NULL if it is not found.

See also:

DDSDomainParticipant::lookup_datawriter_by_name_exp (p. 1213)

6.210 DDSPublisherListener Class Reference

<<*interface*>> *(p. 199)* **DDSListener** (p. 1318) for **DDSPublisher** (p. 1346) status.

Inheritance diagram for DDSPublisherListener::

6.210.1 Detailed Description

<<*interface*>> *(p. 199)* **DDSListener** (p. 1318) for **DDSPublisher** (p. 1346) status.

Entity:

DDSPublisher (p. 1346)

Status:

DDS_LIVELINESS_LOST_STATUS (p. 325), **DDS_LivelinessLostStatus** (p. 777);
DDS_OFFERED_DEADLINE_MISSED_STATUS (p. 323), **DDS_OfferedDeadlineMissedStatus** (p. 803);
DDS_OFFERED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_OfferedIncompatibleQosStatus** (p. 805);
DDS_PUBLICATION_MATCHED_STATUS (p. 325), **DDS_PublicationMatchedStatus** (p. 848);
DDS_RELIABLE_READER_ACTIVITY_CHANGED_STATUS (p. 327), **DDS_ReliableReaderActivityChangedStatus** (p. 869);
DDS_RELIABLE_WRITER_CACHE_CHANGED_STATUS (p. 326), **DDS_ReliableWriterCacheChangedStatus** (p. 871)

See also:

DDSListener (p. 1318)
Status Kinds (p. 317)
Operations Allowed in Listener Callbacks (p. 1320)

6.211 DDSPublisherSeq Class Reference

Declares IDL `sequence` < **DDSPublisher** (p. 1346) > .

6.211.1 Detailed Description

Declares IDL `sequence` < **DDSPublisher** (p. 1346) > .

See also:

FooSeq (p. 1494)

6.212 DDSQueryCondition Class Reference

<<*interface*>> *(p. 199)* These are specialised **DDSReadCondition** (p. 1374) objects that allow the application to also specify a filter on the locally available data.

Inheritance diagram for DDSQueryCondition::

Public Member Functions

- virtual const char * **get_query_expression** ()=0

  Retrieves the query expression.

- virtual **DDS_ReturnCode_t** **get_query_parameters** (**DDS_StringSeq** &query_parameters)=0

  Retrieves the query parameters.

- virtual **DDS_ReturnCode_t** **set_query_parameters** (const **DDS_StringSeq** &query_parameters)=0

  Sets the query parameters.

6.212.1 Detailed Description

<<*interface*>> *(p. 199)* These are specialised **DDSReadCondition** (p. 1374) objects that allow the application to also specify a filter on the locally available data.

Each query condition filter is composed of a **DDSReadCondition** (p. 1374) state filter and a content filter expressed as a `query_expression` and `query_parameters`.

The query (`query_expression`) is similar to an SQL WHERE clause and can be parameterised by arguments that are dynamically changeable by the **set_query_parameters()** (p. 1373) operation.

Two query conditions that have the same `query_expression` will require unique query condition content filters if their `query_paramters` differ. Query conditions that differ only in their state masks will share the same query condition content filter.

Queries and Filters Syntax (p. 208) describes the syntax of `query_expression` and `query_parameters`.

6.212.2 Member Function Documentation

6.212.2.1 virtual const char* DDSQueryCondition::get_query_expression () [pure virtual]

Retrieves the query expression.

6.212.2.2 virtual DDS_ReturnCode_t DDSQueryCondition::get_query_parameters (DDS_StringSeq & *query_parameters*) [pure virtual]

Retrieves the query parameters.

Parameters:

query_parameters <<***inout***>> *(*p. 200*)* the query parameters are returned here. The memory for the strings in this sequence is managed according to the conventions described in **Conventions** (p. 457). In particular, be careful to avoid a situation in which RTI Connext allocates a string on your behalf and you then reuse that string in such a way that RTI Connext believes it to have more memory allocated to it than it actually does.

6.212.2.3 virtual DDS_ReturnCode_t DDSQueryCondition::set_query_parameters (const DDS_StringSeq & *query_parameters*) [pure virtual]

Sets the query parameters.

Parameters:

query_parameters <<***in***>> *(*p. 200*)* the new query parameters

6.213 DDSReadCondition Class Reference

<<*interface*>> *(p. 199)* Conditions specifically dedicated to read operations and attached to one **DDSDataReader** (p. 1087).

Inheritance diagram for DDSReadCondition::

Public Member Functions

- virtual **DDS_SampleStateMask get_sample_state_mask** ()=0

  Retrieves the set of `sample_states` *for the condition.*

- virtual **DDS_ViewStateMask get_view_state_mask** ()=0

  Retrieves the set of `view_states` *for the condition.*

- virtual **DDS_InstanceStateMask get_instance_state_mask** ()=0

  Retrieves the set of `instance_states` *for the condition.*

- virtual **DDSDataReader** * **get_datareader** ()=0

  Returns the ***DDSDataReader*** *(p. 1087) associated with the* ***DDSReadCondition*** *(p. 1374).*

6.213.1 Detailed Description

<<*interface*>> *(p. 199)* Conditions specifically dedicated to read operations and attached to one **DDSDataReader** (p. 1087).

DDSReadCondition (p. 1374) objects allow an application to specify the data samples it is interested in (by specifying the desired `sample_states`, `view_states` as well as `instance_states` in **FooDataReader::read** (p. 1447) and **FooDataReader::take** (p. 1448) variants.

This allows RTI Connext to enable the condition only when suitable information is available. They are to be used in conjunction with a WaitSet as normal conditions.

More than one **DDSReadCondition** (p. 1374) may be attached to the same **DDSDataReader** (p. 1087).

Note: If you are using a ReadCondition simply to detect the presence of new data, consider using a **DDSStatusCondition** (p. 1376) with the DATA_AVAILABLE_STATUS instead, which will perform better in this situation.

6.213.2 Member Function Documentation

6.213.2.1 virtual DDS_SampleStateMask DDSReadCondition::get_sample_state_mask () [pure virtual]

Retrieves the set of `sample_states` for the condition.

6.213.2.2 virtual DDS_ViewStateMask DDSReadCondition::get_view_state_mask () [pure virtual]

Retrieves the set of `view_states` for the condition.

6.213.2.3 virtual DDS_InstanceStateMask DDSReadCondition::get_instance_state_mask () [pure virtual]

Retrieves the set of `instance_states` for the condition.

6.213.2.4 virtual DDSDataReader* DDSReadCondition::get_datareader () [pure virtual]

Returns the **DDSDataReader** (p. 1087) associated with the **DDSReadCondition** (p. 1374).

There is exactly one **DDSDataReader** (p. 1087) assicated with each **DDSReadCondition** (p. 1374).

Returns:

DDSDataReader (p. 1087) associated with the **DDSReadCondition** (p. 1374).

6.214 DDSStatusCondition Class Reference

<<*interface*>> *(p. 199)* A specific **DDSCondition** (p. 1075) that is associated with each **DDSEntity** (p. 1253).

Inheritance diagram for DDSStatusCondition::

Public Member Functions

- virtual **DDS_StatusMask get_enabled_statuses** ()=0

  *Get the list of statuses enabled on an **DDSEntity** (p. 1253).*

- virtual **DDS_ReturnCode_t set_enabled_statuses** (**DDS_StatusMask** mask)=0

  This operation defines the list of communication statuses that determine the `trigger_value` *of the **DDSStatusCondition** (p. 1376).*

- virtual **DDSEntity** * **get_entity** ()=0

  *Get the **DDSEntity** (p. 1253) associated with the **DDSStatusCondition** (p. 1376).*

6.214.1 Detailed Description

<<*interface*>> *(p. 199)* A specific **DDSCondition** (p. 1075) that is associated with each **DDSEntity** (p. 1253).

The `trigger_value` of the **DDSStatusCondition** (p. 1376) depends on the communication status of that entity (e.g., arrival of data, loss of information, etc.), 'filtered' by the set of `enabled_statuses` on the **DDSStatusCondition** (p. 1376).

See also:

Status Kinds (p. 317)
DDSWaitSet (p. 1433), **DDSCondition** (p. 1075)
DDSListener (p. 1318)

6.214.2 Member Function Documentation

6.214.2.1 virtual DDS_StatusMask DDSStatusCondition::get_enabled_statuses () [pure virtual]

Get the list of statuses enabled on an **DDSEntity** (p. 1253).

Returns:

list of enabled statuses.

6.214.2.2 virtual DDS_ReturnCode_t DDSStatusCondition::set_enabled_statuses (DDS_StatusMask *mask*) [pure virtual]

This operation defines the list of communication statuses that determine the `trigger_value` of the **DDSStatusCondition** (p. 1376).

This operation may change the `trigger_value` of the **DDSStatusCondition** (p. 1376).

DDSWaitSet (p. 1433) objects' behavior depends on the changes of the `trigger_value` of their attached conditions. Therefore, any **DDSWaitSet** (p. 1433) to which the **DDSStatusCondition** (p. 1376) is attached is potentially affected by this operation.

If this function is not invoked, the default list of enabled statuses includes all the statuses.

Parameters:

mask *<<in>>* *(p. 200)* the list of enables statuses (see **Status Kinds** (p. 317))

Returns:

One of the **Standard Return Codes** (p. 314)

6.214.2.3 virtual DDSEntity* DDSStatusCondition::get_entity () [pure virtual]

Get the **DDSEntity** (p. 1253) associated with the **DDSStatusCondition** (p. 1376).

There is exactly one **DDSEntity** (p. 1253) assocated with each **DDSStatusCondition** (p. 1376).

Returns:

DDSEntity (p. 1253) associated with the **DDSStatusCondition** (p. 1376).

6.215 DDSStringDataReader Class Reference

<<***interface***>> *(*p. *199)* Instantiates `DataReader` < char* >.

Inheritance diagram for DDSStringDataReader::

Public Member Functions

- virtual **DDS_ReturnCode_t read** (**DDS_StringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  Access a collection of data samples from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t take** (**DDS_StringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples=**DDS_LENGTH_UNLIMITED**, **DDS_SampleStateMask** sample_states=**DDS_ANY_SAMPLE_STATE**, **DDS_ViewStateMask** view_states=**DDS_ANY_VIEW_STATE**, **DDS_InstanceStateMask** instance_states=**DDS_ANY_INSTANCE_STATE**)

  Access a collection of data-samples from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t read_w_condition** (**DDS_StringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  Accesses via ***DDSStringDataReader::read*** *(*p. *1380) the samples that match the criteria specified in the* ***DDSReadCondition*** *(*p. *1374).*

- virtual **DDS_ReturnCode_t take_w_condition** (**DDS_StringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)

  Analogous to ***DDSStringDataReader::read_w_condition*** *(*p. *1381) except it accesses samples via the* ***DDSStringDataReader::take*** *(*p. *1381) operation.*

- virtual **DDS_ReturnCode_t read_next_sample** (char *received_data, **DDS_SampleInfo** &sample_info)

  Copies the next not-previously-accessed data value from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t take_next_sample** (char *received_data, **DDS_SampleInfo** &sample_info)

  *Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).*

- virtual **DDS_ReturnCode_t return_loan** (**DDS_StringSeq** &received_data, **DDS_SampleInfoSeq** &info_seq)

  *Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of* `received_data` *and* `info_seq` *obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).*

Static Public Member Functions

- static **DDSStringDataReader** * **narrow** (**DDSDataReader** *reader)

  *Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSStringDataReader** (p. 1379) pointer.*

6.215.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataReader` < char* >.

See also:

FooDataReader (p. 1444)
DDSDataReader (p. 1087)
String Support (p. 456)

6.215.2 Member Function Documentation

6.215.2.1 virtual DDS_ReturnCode_t DDSStringDataReader::read (DDS_StringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read (p. 1447)

6.215.2.2 virtual DDS_ReturnCode_t DDSStringDataReader::take (DDS_StringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples* = DDS_LENGTH_UNLIMITED, DDS_SampleStateMask *sample_states* = DDS_ANY_SAMPLE_STATE, DDS_ViewStateMask *view_states* = DDS_ANY_VIEW_STATE, DDS_InstanceStateMask *instance_states* = DDS_ANY_INSTANCE_STATE) [virtual]

Access a collection of data-samples from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take (p. 1448)

6.215.2.3 virtual DDS_ReturnCode_t DDSStringDataReader::read_w_condition (DDS_StringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Accesses via **DDSStringDataReader::read** (p. 1380) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

See also:

FooDataReader::read_w_condition (p. 1454)

6.215.2.4 virtual DDS_ReturnCode_t DDSStringDataReader::take_w_condition (DDS_StringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [virtual]

Analogous to **DDSStringDataReader::read_w_condition** (p. 1381) except it accesses samples via the **DDSStringDataReader::take** (p. 1381) operation.

See also:

FooDataReader::take_w_condition (p. 1456)

6.215.2.5 virtual DDS_ReturnCode_t DDSStringDataReader::read_next_sample (char * *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::read_next_sample (p. 1457)

6.215.2.6 virtual DDS_ReturnCode_t DDSStringDataReader::take_next_sample (char * *received_data*, DDS_SampleInfo & *sample_info*) [virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

See also:

FooDataReader::take_next_sample (p. 1458)

6.215.2.7 virtual DDS_ReturnCode_t DDSStringDataReader::return_loan (DDS_StringSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*) [virtual]

Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of `received_data` and `info_seq` obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).

See also:

FooDataReader::return_loan (p. 1471)

6.215.2.8 static DDSStringDataReader* DDSStringDataReader::narrow (DDSDataReader * *reader*) [static]

Narrow the given **DDSDataReader** (p. 1087) pointer to a **DDSStringDataReader** (p. 1379) pointer.

See also:

FooDataReader::narrow (p. 1447)

6.216 DDSStringDataWriter Class Reference

<<***interface***>> *(p. 199)* Instantiates `DataWriter` < char* >.

Inheritance diagram for DDSStringDataWriter::

Public Member Functions

- virtual **DDS_ReturnCode_t write** (const char *instance_data, const **DDS_InstanceHandle_t** &handle)

  Modifies the value of a string data instance.

- virtual **DDS_ReturnCode_t write_w_timestamp** (const char *instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)

  Performs the same function as ***DDSStringDataWriter::write*** *(p. 1384) except that it also provides the value for the* `source_timestamp`*.*

- virtual **DDS_ReturnCode_t write_w_params** (const char *instance_data, **DDS_WriteParams_t** &params)

  Performs the same function as ***DDSStringDataWriter::write*** *(p. 1384) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.*

Static Public Member Functions

- static **DDSStringDataWriter** * **narrow** (**DDSDataWriter** *writer)

  Narrow the given ***DDSDataWriter*** *(p. 1113) pointer to a* ***DDSStringDataWriter*** *(p. 1383) pointer.*

6.216.1 Detailed Description

<<***interface***>> *(p. 199)* Instantiates `DataWriter` < char* >.

See also:

FooDataWriter (p. 1475)
DDSDataWriter (p. 1113)
String Support (p. 456)

6.216.2 Member Function Documentation

6.216.2.1 static DDSStringDataWriter* DDSStringDataWriter::narrow (DDSDataWriter * *writer*) [static]

Narrow the given **DDSDataWriter** (p. 1113) pointer to a **DDSStringDataWriter** (p. 1383) pointer.

See also:

FooDataWriter::narrow (p. 1477)

6.216.2.2 virtual DDS_ReturnCode_t DDSStringDataWriter::write (const char * *instance_data*, const DDS_InstanceHandle_t & *handle*) [virtual]

Modifies the value of a string data instance.

See also:

FooDataWriter::write (p. 1484)

6.216.2.3 virtual DDS_ReturnCode_t DDSStringDataWriter::write_w_timestamp (const char * *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [virtual]

Performs the same function as **DDSStringDataWriter::write** (p. 1384) except that it also provides the value for the `source_timestamp`.

See also:

FooDataWriter::write_w_timestamp (p. 1486)

6.216.2.4 virtual DDS_ReturnCode_t DDSStringDataWriter::write_w_params (const char * *instance_data*, DDS_WriteParams_t & *params*) [virtual]

Performs the same function as **DDSStringDataWriter::write** (p. 1384) except that it also allows specification of the instance handle, source timestamp, publication priority, and cookie.

See also:

FooDataWriter::write_w_params (p. 1487)

6.217 DDSStringTypeSupport Class Reference

<<***interface***>> *(p. 199)* String type support.

Inheritance diagram for DDSStringTypeSupport::

Static Public Member Functions

- static **DDS_ReturnCode_t register_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::String")

  Allows an application to communicate to RTI Connext the existence of the char data type.*

- static **DDS_ReturnCode_t unregister_type** (**DDSDomainParticipant** *participant, const char *type_name="DDS::String")

  Allows an application to unregister the char data type from RTI Connext. After calling unregister_type, no further communication using this type is possible.*

- static const char * **get_type_name** ()

  Get the default name for the char type.*

- static void **print_data** (const char *a_data)

  <<**eXtension**>> (p. 199) *Print value of data type to standard out.*

6.217.1 Detailed Description

<<***interface***>> *(p. 199)* String type support.

6.217.2 Member Function Documentation

6.217.2.1 static DDS_ReturnCode_t DDSStringTypeSupport::register_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::String") [static]

Allows an application to communicate to RTI Connext the existence of the char* data type.

By default, The char* built-in type is automatically registered when a DomainParticipant is created using the type_name returned by **DDSStringTypeSupport::get_type_name** (p. 1388). Therefore, the usage of this function is op-

tional and it is only required when the automatic built-in type registration is disabled using the participant property "dds.builtin_type.auto_register".

This method can also be used to register the same **DDSStringTypeSupport** (p. 1386) with a **DDSDomainParticipant** (p. 1139) using different values for the type_name.

If `register_type` is called multiple times with the same **DDSDomainParticipant** (p. 1139) and `type_name`, the second (and subsequent) registrations are ignored by the operation.

Parameters:

participant *<<in>>* *(p. 200)* the **DDSDomainParticipant** (p. 1139) to register the data type char* with. Cannot be NULL.

type_name *<<in>>* *(p. 200)* the type name under with the data type char* is registered with the participant; this type name is used when creating a new **DDSTopic** (p. 1419). (See **DDSDomainParticipant::create_topic** (p. 1175).) The name may not be NULL or longer than 255 characters.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to register a type. Subsequent calls are thread safe.

See also:

DDSDomainParticipant::create_topic (p. 1175)

6.217.2.2 static DDS_ReturnCode_t DDSStringTypeSupport::unregister_type (DDSDomainParticipant * *participant*, const char * *type_name* = "DDS::String") [static]

Allows an application to unregister the char* data type from RTI Connext. After calling unregister_type, no further communication using this type is possible.

Precondition:

The char* type with `type_name` is registered with the participant and all **DDSTopic** (p. 1419) objects referencing the type have been destroyed.

If the type is not registered with the participant, or if any **DDSTopic** (p. 1419) is associated with the type, the operation will fail with **DDS_RETCODE_ERROR** (p. 315).

Postcondition:

All information about the type is removed from RTI Connext. No further communication using this type is possible.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to unregister the data type char* from. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type char* is registered with the participant. The name should match a name that has been previously used to register a type with the participant. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_BAD_PARAMETER** (p. 315) or **DDS_RETCODE_ERROR** (p. 315)

MT Safety:

SAFE.

See also:

DDSStringTypeSupport::register_type (p. 1386)

6.217.2.3 static const char* DDSStringTypeSupport::get_type_name () [static]

Get the default name for the char* type.

Can be used for calling **DDSStringTypeSupport::register_type** (p. 1386) or creating **DDSTopic** (p. 1419).

Returns:

default name for the char* type.

See also:

DDSStringTypeSupport::register_type (p. 1386)
DDSDomainParticipant::create_topic (p. 1175)

6.217.2.4 static void DDSStringTypeSupport::print_data (const char * *a_data*) [static]

<<*eXtension*>> *(*p. *199)* Print value of data type to standard out.

The *generated* implementation of the operation knows how to print value of a data type.

Parameters:

a_data <<*in*>> *(*p. *200)* String to be printed.

6.218 DDSSubscriber Class Reference

<<***interface***>> *(p. 199)* A subscriber is the object responsible for actually receiving data from a subscription.

Inheritance diagram for DDSSubscriber::

Public Member Functions

- virtual **DDS_ReturnCode_t get_default_datareader_qos** (**DDS_DataReaderQos** &qos)=0

  Copies the default ***DDS_DataReaderQos*** *(p. 515) values into the provided* ***DDS_DataReaderQos*** *(p. 515) instance.*

- virtual **DDS_ReturnCode_t set_default_datareader_qos** (const **DDS_DataReaderQos** &qos)=0

  Sets the default ***DDS_DataReaderQos*** *(p. 515) values for this subscriber.*

- virtual **DDS_ReturnCode_t set_default_datareader_qos_with_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Set the default* ***DDS_DataReaderQos*** *(p. 515) values for this subscriber based on the input XML QoS profile.*

- virtual **DDS_ReturnCode_t set_default_library** (const char *library_name)=0

  <<**eXtension**>> (p. 199) *Sets the default XML library for a* ***DDSSubscriber*** *(p. 1390).*

- virtual const char * **get_default_library** ()=0

  <<**eXtension**>> (p. 199) *Gets the default XML library associated with a* ***DDSSubscriber*** *(p. 1390).*

- virtual **DDS_ReturnCode_t set_default_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Sets the default XML profile for a* ***DDSSubscriber*** *(p. 1390).*

- virtual const char * **get_default_profile** ()=0

  <<**eXtension**>> (p. 199) *Gets the default XML profile associated with a* ***DDSSubscriber*** *(p. 1390).*

- virtual const char * **get_default_profile_library** ()=0

<<**eXtension**>> (p. 199) *Gets the library where the default XML QoS profile is contained for a* ***DDSSubscriber*** *(*p. *1390).*

ˆ virtual **DDSDataReader** ∗ **create_datareader** (**DDSTopicDescription** ∗topic, const **DDS_DataReaderQos** &qos, **DDSDataReaderListener** ∗listener, **DDS_StatusMask** mask)=0

Creates a ***DDSDataReader*** *(*p. *1087) that will be attached and belong to the* ***DDSSubscriber*** *(*p. *1390).*

ˆ virtual **DDSDataReader** ∗ **create_datareader_with_profile** (**DDSTopicDescription** ∗topic, const char ∗library_name, const char ∗profile_name, **DDSDataReaderListener** ∗listener, **DDS_StatusMask** mask)=0

<<**eXtension**>> (p. 199) *Creates a* ***DDSDataReader*** *(*p. *1087) object using the* ***DDS_DataReaderQos*** *(*p. *515) associated with the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t delete_datareader** (**DDSDataReader** ∗a_datareader)=0

Deletes a ***DDSDataReader*** *(*p. *1087) that belongs to the* ***DDSSubscriber*** *(*p. *1390).*

ˆ virtual **DDS_ReturnCode_t delete_contained_entities** ()=0

Deletes all the entities that were created by means of the "create" operation on the ***DDSSubscriber*** *(*p. *1390).*

ˆ virtual **DDSDataReader** ∗ **lookup_datareader** (const char ∗topic_name)=0

Retrieves an existing ***DDSDataReader*** *(*p. *1087).*

ˆ virtual **DDS_ReturnCode_t begin_access** ()=0

Indicates that the application is about to access the data samples in any of the ***DDSDataReader*** *(*p. *1087) objects attached to the* ***DDSSubscriber*** *(*p. *1390).*

ˆ virtual **DDS_ReturnCode_t end_access** ()=0

Indicates that the application has finished accessing the data samples in ***DDSDataReader*** *(*p. *1087) objects managed by the* ***DDSSubscriber*** *(*p. *1390).*

ˆ virtual **DDS_ReturnCode_t get_datareaders** (**DDSDataReaderSeq** &readers, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

Allows the application to access the ***DDSDataReader*** *(*p. *1087) objects that contain samples with the specified* `sample_states`, `view_states` *and* `instance_states`.

ˆ virtual **DDS_ReturnCode_t get_all_datareaders** (**DDSDataReaderSeq** &readers)=0

Retrieve all the DataReaders created from this Subscriber.

ˆ virtual **DDS_ReturnCode_t notify_datareaders** ()=0

Invokes the operation ***DDSDataReaderListener::on_data_available()*** *(*p. *1110) on the* ***DDSDataReaderListener*** *(*p. *1108) objects attached to contained* ***DDSDataReader*** *(*p. *1087) entities with* ***DDS_DATA_AVAILABLE_STATUS*** *(*p. *324) that is considered changed as described in* ***Changes in read communication status*** *(*p. *320).

ˆ virtual **DDSDomainParticipant** ∗ **get_participant** ()=0

Returns the ***DDSDomainParticipant*** *(*p. *1139) to which the* ***DDSSubscriber*** *(*p. *1390) belongs.

ˆ virtual **DDS_ReturnCode_t copy_from_topic_qos** (**DDS_DataReaderQos** &datareader_qos, const **DDS_TopicQos** &topic_qos)=0

Copies the policies in the ***DDS_TopicQos*** *(*p. *965) to the corresponding policies in the* ***DDS_DataReaderQos*** *(*p. *515).

ˆ virtual **DDS_ReturnCode_t set_qos** (const **DDS_SubscriberQos** &qos)=0

Sets the subscriber QoS.

ˆ virtual **DDS_ReturnCode_t set_qos_with_profile** (const char ∗library_name, const char ∗profile_name)=0

<<**eXtension**>> (p. 199) *Change the QoS of this subscriber using the input XML QoS profile.*

ˆ virtual **DDS_ReturnCode_t get_qos** (**DDS_SubscriberQos** &qos)=0

Gets the subscriber QoS.

ˆ virtual **DDS_ReturnCode_t set_listener** (**DDSSubscriberListener** ∗l, **DDS_StatusMask** mask=DDS_STATUS_MASK_ALL)=0

Sets the subscriber listener.

ˆ virtual **DDSSubscriberListener** ∗ **get_listener** ()=0

Get the subscriber listener.

- virtual **DDSDataReader** * **lookup_datareader_by_name_exp** (const char *datareader_name)=0

  <<**experimental**>> (p. 199) <<**eXtension**>> (p. 199) *Retrieves a **DDSDataReader** (p. 1087) contained within the **DDSSubscriber** (p. 1390) the **DDSDataReader** (p. 1087) entity name.*

6.218.1 Detailed Description

<<***interface***>> *(p. 199)* A subscriber is the object responsible for actually receiving data from a subscription.

QoS:

DDS_SubscriberQos (p. 934)

Status:

DDS_DATA_ON_READERS_STATUS (p. 324)

Listener:

DDSSubscriberListener (p. 1414)

A subscriber acts on the behalf of one or several **DDSDataReader** (p. 1087) objects that are related to it. When it receives data (from the other parts of the system), it builds the list of concerned **DDSDataReader** (p. 1087) objects and then indicates to the application that data is available through its listener or by enabling related conditions.

The application can access the list of concerned **DDSDataReader** (p. 1087) objects through the operation **get_datareaders()** (p. 1407) and then access the data available through operations on the **DDSDataReader** (p. 1087).

The following operations may be called even if the **DDSSubscriber** (p. 1390) is not enabled. Other operations will the value **DDS_RETCODE_NOT_ENABLED** (p. 315) if called on a disabled **DDSSubscriber** (p. 1390):

- The base-class operations **DDSSubscriber::set_qos** (p. 1410), **DDSSubscriber::set_qos_with_profile** (p. 1410), **DDSSubscriber::get_qos** (p. 1411), **DDSSubscriber::set_listener** (p. 1411), **DDSSubscriber::get_listener** (p. 1412), **DDSEntity::enable** (p. 1256), **DDSEntity::get_statuscondition** (p. 1257), **DDSEntity::get_status_changes** (p. 1257)

- ^ **DDSSubscriber::create_datareader** (p. 1399), **DDSSubscriber::create_datareader_with_profile** (p. 1401), **DDSSubscriber::delete_datareader** (p. 1403), **DDSSubscriber::delete_contained_entities** (p. 1403), **DDSSubscriber::set_default_datareader_qos** (p. 1395), **DDSSubscriber::set_default_datareader_qos_with_profile** (p. 1396), **DDSSubscriber::get_default_datareader_qos** (p. 1394), **DDSSubscriber::set_default_library** (p. 1397), **DDSSubscriber::set_default_profile** (p. 1398)

All operations except for the base-class operations **set_qos()** (p. 1410), **set_qos_with_profile()** (p. 1410), **get_qos()** (p. 1411), **set_listener()** (p. 1411), **get_listener()** (p. 1412), **enable()** (p. 1256) and **create_datareader()** (p. 1399) may fail with **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_subscriber.cxx.

6.218.2 Member Function Documentation

6.218.2.1 virtual DDS_ReturnCode_t DDSSubscriber::get_default_datareader_qos (DDS_DataReaderQos & *qos*) [pure virtual]

Copies the default **DDS_DataReaderQos** (p. 515) values into the provided **DDS_DataReaderQos** (p. 515) instance.

The retrieved `qos` will match the set of values specified on the last successful call to **DDSSubscriber::set_default_datareader_qos** (p. 1395), or **DDSSubscriber::set_default_datareader_qos_with_profile** (p. 1396), or else, if the call was never made, the default values from is owning **DDSDomainParticipant** (p. 1139).

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

MT Safety:

UNSAFE. It is not safe to retrieve the default QoS value from a subscriber while another thread may be simultaneously calling **DDSSubscriber::set_default_datareader_qos** (p. 1395)

Parameters:

qos *<<inout>>* *(p. 200)* **DDS_DataReaderQos** (p. 515) to be filled-up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_DATAREADER_QOS_DEFAULT (p. 99)
DDSSubscriber::create_datareader (p. 1399)

6.218.2.2 virtual DDS_ReturnCode_t DDSSubscriber::set_default_datareader_qos (const DDS_DataReaderQos & *qos*) [pure virtual]

Sets the default **DDS_DataReaderQos** (p. 515) values for this subscriber.

This call causes the default values inherited from the owning **DDSDomainParticipant** (p. 1139) to be overridden.

This default value will be used for newly created **DDSDataReader** (p. 1087) if **DDS_DATAREADER_QOS_DEFAULT** (p. 99) is specified as the `qos` parameter when **DDSSubscriber::create_datareader** (p. 1399) is called.

Precondition:

The specified QoS policies must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default QoS value from a subscriber while another thread may be simultaneously calling **DDSSubscriber::set_default_datareader_qos** (p. 1395), **DDSSubscriber::get_default_datareader_qos** (p. 1394) or calling **DDSSubscriber::create_datareader** (p. 1399) with **DDS_DATAREADER_QOS_DEFAULT** (p. 99) as the `qos` parameter.

Parameters:

qos <<*in*>> *(p. 200)* The default **DDS_DataReaderQos** (p. 515) to be set to. The special value **DDS_DATAREADER_QOS_DEFAULT** (p. 99) may be passed as `qos` to indicate that the default QoS should be reset back to the initial values the factory would used if **DDSSubscriber::set_default_datareader_qos** (p. 1395) had never been called.

Returns:

One of the **Standard Return Codes** (p. 314), or or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

6.218.2.3 virtual DDS_ReturnCode_t DDSSubscriber::set_default_datareader_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> *(p. 199)* Set the default **DDS_DataReaderQos** (p. 515) values for this subscriber based on the input XML QoS profile.

This default value will be used for newly created **DDSDataReader** (p. 1087) if **DDS_DATAREADER_QOS_DEFAULT** (p. 99) is specified as the `qos` parameter when **DDSSubscriber::create_datareader** (p. 1399) is called.

Precondition:

The **DDS_DataReaderQos** (p. 515) contained in the specified XML QoS profile must be consistent, or else the operation will have no effect and fail with **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

MT Safety:

UNSAFE. It is not safe to set the default QoS value from a **DDSSubscriber** (p. 1390) while another thread may be simultaneously calling **DDSSubscriber::set_default_datareader_qos** (p. 1395), **DDSSubscriber::get_default_datareader_qos** (p. 1394) or calling **DDSSubscriber::create_datareader** (p. 1399) with **DDS_DATAREADER_QOS_DEFAULT** (p. 99) as the `qos` parameter.

Parameters:

library_name *<<in>>* *(p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSSubscriber::set_default_library** (p. 1397)).

profile_name *<<in>>* *(p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSSubscriber::set_default_profile** (p. 1398)).

If the input profile cannot be found the method fails with **DDS_RETCODE_ERROR** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315)

See also:

DDS_DATAREADER_QOS_DEFAULT (p. 99)
DDSSubscriber::create_datareader_with_profile (p. 1401)

6.218.2.4 virtual DDS_ReturnCode_t DDSSubscriber::set_default_library (const char * *library_name*) [pure virtual]

<<eXtension>> (p. 199) Sets the default XML library for a **DDSSubscriber** (p. 1390).

This method specifies the library that will be used as the default the next time a default library is needed during a call to one of this Subscriber's operations.

Any API requiring a library_name as a parameter can use null to refer to the default library.

If the default library is not set, the **DDSSubscriber** (p. 1390) inherits the default from the **DDSDomainParticipant** (p. 1139) (see **DDSDomainParticipant::set_default_library** (p. 1159)).

Parameters:

library_name *<<in>> (p. 200)* Library name. If library_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSSubscriber::get_default_library (p. 1397)

6.218.2.5 virtual const char* DDSSubscriber::get_default_library () [pure virtual]

<<eXtension>> (p. 199) Gets the default XML library associated with a **DDSSubscriber** (p. 1390).

Returns:

The default library or null if the default library was not set.

See also:

DDSSubscriber::set_default_library (p. 1397)

6.218.2.6 virtual DDS_ReturnCode_t DDSSubscriber::set_default_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> (p. 199) Sets the default XML profile for a **DDSSubscriber** (p. 1390).

This method specifies the profile that will be used as the default the next time a default Subscriber profile is needed during a call to one of this Subscriber's operations. When calling a **DDSSubscriber** (p. 1390) method that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

If the default profile is not set, the **DDSSubscriber** (p. 1390) inherits the default from the **DDSDomainParticipant** (p. 1139) (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

This method does not set the default QoS for **DDSDataReader** (p. 1087) objects created by this **DDSSubscriber** (p. 1390); for this functionality, use **DDSSubscriber::set_default_datareader_qos_with_profile** (p. 1396) (you may pass in NULL after having called **set_default_profile()** (p. 1398)).

This method does not set the default QoS for newly created Subscribers; for this functionality, use **DDSDomainParticipant::set_default_subscriber_qos_with_profile** (p. 1168).

Parameters:

library_name *<<in>> (p. 200)* The library name containing the profile.

profile_name *<<in>> (p. 200)* The profile name. If profile_name is null any previous default is unset.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDSSubscriber::get_default_profile (p. 1398)
DDSSubscriber::get_default_profile_library (p. 1399)

6.218.2.7 virtual const char* DDSSubscriber::get_default_profile () [pure virtual]

<<eXtension>> (p. 199) Gets the default XML profile associated with a **DDSSubscriber** (p. 1390).

Returns:

The default profile or null if the default profile was not set.

See also:

DDSSubscriber::set_default_profile (p. 1398)

6.218.2.8 virtual const char* DDSSubscriber::get_default_profile_library () [pure virtual]

<<*eXtension*>> *(p. 199)* Gets the library where the default XML QoS profile is contained for a **DDSSubscriber** (p. 1390).

The default profile library is automatically set when **DDSSubscriber::set_default_profile** (p. 1398) is called.

This library can be different than the **DDSSubscriber** (p. 1390) default library (see **DDSSubscriber::get_default_library** (p. 1397)).

Returns:

The default profile library or null if the default profile was not set.

See also:

DDSSubscriber::set_default_profile (p. 1398)

6.218.2.9 virtual DDSDataReader* DDSSubscriber::create_datareader (DDSTopicDescription * *topic*, const DDS_DataReaderQos & *qos*, DDSDataReaderListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

Creates a **DDSDataReader** (p. 1087) that will be attached and belong to the **DDSSubscriber** (p. 1390).

For each application-defined type `Foo` (p. 1443), there is an implied, auto-generated class `FooDataReader` (p. 1444) (an incarnation of **FooDataReader** (p. 1444)) that extends **DDSDataReader** (p. 1087) and contains the operations to read data of type `Foo` (p. 1443).

Note that a common application pattern to construct the QoS for the **DDSDataReader** (p. 1087) is to:

- Retrieve the QoS policies on the associated **DDSTopic** (p. 1419) by means of the **DDSTopic::get_qos** (p. 1423) operation.

- Retrieve the default **DDSDataReader** (p. 1087) qos by means of the **DDSSubscriber::get_default_datareader_qos** (p. 1394) operation.
- Combine those two QoS policies (for example, using **DDSSubscriber::copy_from_topic_qos** (p. 1409)) and selectively modify policies as desired
- Use the resulting QoS policies to construct the **DDSDataReader** (p. 1087).

When a **DDSDataReader** (p. 1087) is created, only those transports already registered are available to the **DDSDataReader** (p. 1087). See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

MT Safety:

UNSAFE. If **DDS_DATAREADER_QOS_DEFAULT** (p. 99) is used for the `qos` parameter, it is not safe to create the datareader while another thread may be simultaneously calling **DDSSubscriber::set_default_datareader_qos** (p. 1395).

Precondition:

If subscriber is enabled, the topic must be enabled. If it is not, this operation will fail and no **DDSDataReader** (p. 1087) will be created.
The given **DDSTopicDescription** (p. 1427) must have been created from the same participant as this subscriber. If it was created from a different participant, this method will return NULL.
If `qos` is **DDS_DATAREADER_QOS_USE_TOPIC_QOS** (p. 99), `topic` cannot be **DDSMultiTopic** (p. 1322), or else this method will return NULL.

Parameters:

topic *<<in>> (p. 200)* The **DDSTopicDescription** (p. 1427) that the **DDSDataReader** (p. 1087) will be associated with. Cannot be NULL.

qos *<<in>> (p. 200)* The qos of the **DDSDataReader** (p. 1087). The special value **DDS_DATAREADER_QOS_DEFAULT** (p. 99) can be used to indicate that the **DDSDataReader** (p. 1087) should be created with the default **DDS_DataReaderQos** (p. 515) set in the **DDSSubscriber** (p. 1390). If **DDSTopicDescription** (p. 1427) is of type **DDSTopic** (p. 1419) or **DDSContentFilteredTopic** (p. 1081), the special value **DDS_DATAREADER_QOS_USE_TOPIC_QOS** (p. 99) can be used to indicate that the **DDSDataReader** (p. 1087) should be created with the combination of

the default **DDS_DataReaderQos** (p. 515) set on the **DDSSubscriber** (p. 1390) and the **DDS_TopicQos** (p. 965) (in the case of a **DDSContentFilteredTopic** (p. 1081), the **DDS_TopicQos** (p. 965) of the related **DDSTopic** (p. 1419)). if **DDS_DATAREADER_QOS_USE_TOPIC_QOS** (p. 99) is used, `topic` cannot be a **DDSMultiTopic** (p. 1322).

listener *<<in>> (p. 200)* The listener of the **DDSDataReader** (p. 1087).

mask *<<in>> (p. 200)*. Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataReader** (p. 1087) of a derived class specific to the data-type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataReader (p. 1444)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataReaderQos (p. 515) for rules on consistency among QoS
DDSSubscriber::create_datareader_with_profile (p. 1401)
DDSSubscriber::get_default_datareader_qos (p. 1394)
DDSTopic::set_qos (p. 1421)
DDSSubscriber::copy_from_topic_qos (p. 1409)
DDSDataReader::set_listener (p. 1104)

Examples:

HelloWorld_subscriber.cxx.

6.218.2.10 virtual DDSDataReader* DDSSubscriber::create_datareader_with_profile (DDSTopicDescription * *topic*, const char * *library_name*, const char * *profile_name*, DDSDataReaderListener * *listener*, DDS_StatusMask *mask*) [pure virtual]

<<eXtension>> (p. 199) Creates a **DDSDataReader** (p. 1087) object using the **DDS_DataReaderQos** (p. 515) associated with the input XML QoS profile.

The **DDSDataReader** (p. 1087) will be attached and belong to the **DDSSubscriber** (p. 1390).

For each application-defined type `Foo` (p. 1443), there is an implied, auto-generated class `FooDataReader` (p. 1444) (an incarnation of **FooDataReader**

(p. 1444)) that extends **DDSDataReader** (p. 1087) and contains the operations to read data of type `Foo` (p. 1443).

When a **DDSDataReader** (p. 1087) is created, only those transports already registered are available to the **DDSDataReader** (p. 1087). See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

Precondition:

If subscriber is enabled, the topic must be enabled. If it is not, this operation will fail and no **DDSDataReader** (p. 1087) will be created.
The given **DDSTopicDescription** (p. 1427) must have been created from the same participant as this subscriber. If it was created from a different participant, this method will return NULL.

Parameters:

topic *<<in>> (p. 200)* The **DDSTopicDescription** (p. 1427) that the **DDSDataReader** (p. 1087) will be associated with. Cannot be NULL.

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSSubscriber::set_default_library** (p. 1397)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSSubscriber::set_default_profile** (p. 1398)).

listener *<<in>> (p. 200)* The listener of the **DDSDataReader** (p. 1087).

mask *<<in>> (p. 200).* Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

A **DDSDataReader** (p. 1087) of a derived class specific to the data-type associated with the **DDSTopic** (p. 1419) or NULL if an error occurred.

See also:

FooDataReader (p. 1444)
Specifying QoS on entities (p. 338) for information on setting QoS before entity creation
DDS_DataReaderQos (p. 515) for rules on consistency among QoS
DDS_DATAREADER_QOS_DEFAULT (p. 99)
DDS_DATAREADER_QOS_USE_TOPIC_QOS (p. 99)
DDSSubscriber::create_datareader (p. 1399)
DDSSubscriber::get_default_datareader_qos (p. 1394)

DDSTopic::set_qos (p. 1421)
DDSSubscriber::copy_from_topic_qos (p. 1409)
DDSDataReader::set_listener (p. 1104)

6.218.2.11 virtual DDS_ReturnCode_t DDSSubscriber::delete_datareader (DDSDataReader * *a_datareader*) [pure virtual]

Deletes a **DDSDataReader** (p. 1087) that belongs to the **DDSSubscriber** (p. 1390).

Precondition:

If the **DDSDataReader** (p. 1087) does not belong to the **DDSSubscriber** (p. 1390), or if ther are any existing **DDSReadCondition** (p. 1374) or **DDSQueryCondition** (p. 1372) objects that are attached to the **DDSDataReader** (p. 1087), or if there are outstanding loans on samples (as a result of a call to read(), take(), or one of the variants thereof), the operation fails with the error **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Postcondition:

Listener installed on the **DDSDataReader** (p. 1087) will not be called after this method completes successfully.

Parameters:

a_datareader *<<in>> (p. 200)* The **DDSDataReader** (p. 1087) to be deleted.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

6.218.2.12 virtual DDS_ReturnCode_t DDSSubscriber::delete_contained_entities () [pure virtual]

Deletes all the entities that were created by means of the "create" operation on the **DDSSubscriber** (p. 1390).

Deletes all contained **DDSDataReader** (p. 1087) objects. This pattern is applied recursively. In this manner, the operation **DDSSubscriber::delete_contained_entities** (p. 1403) on the **DDSSubscriber** (p. 1390) will end up

deleting all the entities recursively contained in the **DDSSubscriber** (p. 1390), that is also the **DDSQueryCondition** (p. 1372) and **DDSReadCondition** (p. 1374) objects belonging to the contained **DDSDataReader** (p. 1087).

The operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) if any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained **DDSDataReader** (p. 1087) cannot be deleted because the application has called a **FooDataReader::read** (p. 1447) or **FooDataReader::take** (p. 1448) operation and has not called the corresponding **FooDataReader::return_loan** (p. 1471) operation to return the loaned samples.

Once **DDSSubscriber::delete_contained_entities** (p. 1403) completes successfully, the application may delete the **DDSSubscriber** (p. 1390), knowing that it has no contained **DDSDataReader** (p. 1087) objects.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315)

6.218.2.13 virtual DDSDataReader* DDSSubscriber::lookup_datareader (const char * *topic_name*) [pure virtual]

Retrieves an existing **DDSDataReader** (p. 1087).

Use this operation on the built-in **DDSSubscriber** (p. 1390) (**Built-in Topics** (p. 42)) to access the built-in **DDSDataReader** (p. 1087) entities for the built-in topics.

The built-in **DDSDataReader** (p. 1087) is created when this operation is called on a built-in topic for the first time. The built-in **DDSDataReader** (p. 1087) is deleted automatically when the **DDSDomainParticipant** (p. 1139) is deleted.

To ensure that builtin **DDSDataReader** (p. 1087) entities receive all the discovery traffic, it is suggested that you lookup the builtin **DDSDataReader** (p. 1087) before the **DDSDomainParticipant** (p. 1139) is enabled. Looking up builtin **DDSDataReader** (p. 1087) may implicitly register builtin transports due to creation of **DDSDataReader** (p. 1087) (see **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered). Therefore, if you are want to modify builtin transport properties, do so *before* using this operation.

Therefore the suggested sequence when looking up builtin DataReaders is:

- Create a disabled **DDSDomainParticipant** (p. 1139).
- (optional) Modify builtin transport properties

- Call **DDSDomainParticipant::get_builtin_subscriber()** (p. 1186).
- Call **lookup_datareader()** (p. 1404).
- Call **enable()** (p. 1256) on the DomainParticipant.

Parameters:

topic_name *<<in>>* *(p. 200)* Name of the **DDSTopicDescription** (p. 1427) that the retrieved **DDSDataReader** (p. 1087) is attached to. Cannot be NULL.

Returns:

A **DDSDataReader** (p. 1087) that belongs to the **DDSSubscriber** (p. 1390) attached to the **DDSTopicDescription** (p. 1427) with `topic_name`. If no such **DDSDataReader** (p. 1087) exists, this operation returns NULL.

The returned **DDSDataReader** (p. 1087) may be enabled or disabled.

If more than one **DDSDataReader** (p. 1087) is attached to the **DDSSubscriber** (p. 1390), this operation may return any one of them.

MT Safety:

UNSAFE. It is not safe to lookup a **DDSDataReader** (p. 1087) in one thread while another thread is simultaneously creating or destroying that **DDSDataReader** (p. 1087).

6.218.2.14 virtual DDS_ReturnCode_t DDSSubscriber::begin_access () [pure virtual]

Indicates that the application is about to access the data samples in any of the **DDSDataReader** (p. 1087) objects attached to the **DDSSubscriber** (p. 1390).

If the **DDS_PresentationQosPolicy::access_scope** (p. 826) of the **DDSSubscriber** (p. 1390) is **DDS_GROUP_PRESENTATION_QOS** (p. 352) or **DDS_HIGHEST_OFFERED_PRESENTATION_QOS** (p. 352) and **DDS_PresentationQosPolicy::ordered_access** (p. 827) is **DDS_BOOLEAN_TRUE** (p. 298), the application is required to use this operation to access the samples in order across DataWriters of the same group (**DDSPublisher** (p. 1346) with **DDS_PresentationQosPolicy::access_scope** (p. 826) set to **DDS_GROUP_PRESENTATION_QOS** (p. 352)).

In the above case, the operation **begin_access()** (p. 1405) must be called prior to calling any of the sample-accessing operations, **DDSSubscriber::get_datareaders** (p. 1407) on the **DDSSubscriber** (p. 1390), and

FooDataReader::read (p. 1447), **FooDataReader::take** (p. 1448), **FooDataReader::read_w_condition** (p. 1454), and **FooDataReader::take_w_condition** (p. 1456) on any **DDSDataReader** (p. 1087).

Once the application has finished accessing the data samples, it must call **DDSSubscriber::end_access** (p. 1406).

The application is not required to call **begin_access()** (p. 1405) / **end_access()** (p. 1406) to access the samples in order if the **PRESENTATION** (p. 351) policy in the **DDSPublisher** (p. 1346) has **DDS_PresentationQosPolicy::access_scope** (p. 826) set to something other than **DDS_GROUP_PRESENTATION_QOS** (p. 352). In this case, calling **begin_access()** (p. 1405) / **end_access()** (p. 1406) is not considered an error and has no effect.

Calls to **begin_access()** (p. 1405) / **end_access()** (p. 1406) may be nested and must be balanced.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

Access to data samples (p. 95)
DDSSubscriber::get_datareaders (p. 1407)
PRESENTATION (p. 351)

6.218.2.15 virtual DDS_ReturnCode_t DDSSubscriber::end_access () [pure virtual]

Indicates that the application has finished accessing the data samples in **DDSDataReader** (p. 1087) objects managed by the **DDSSubscriber** (p. 1390).

This operation must be used to close a corresponding **begin_access()** (p. 1405).

This call must close a previous call to **DDSSubscriber::begin_access()** (p. 1405), otherwise the operation will fail with the error **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.218.2.16 virtual DDS_ReturnCode_t DDSSubscriber::get_datareaders (DDSDataReaderSeq & *readers*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Allows the application to access the **DDSDataReader** (p. 1087) objects that contain samples with the specified `sample_states`, `view_states` and `instance_states`.

If the application is outside a **begin_access()** (p. 1405)/end_access() block, or if the **DDS_PresentationQosPolicy::access_scope** (p. 826) of the **DDSSubscriber** (p. 1390) is **DDS_INSTANCE_PRESENTATION_QOS** (p. 352) or **DDS_TOPIC_PRESENTATION_QOS** (p. 352), or if the **DDS_PresentationQosPolicy::ordered_access** (p. 827) of the **DDSSubscriber** (p. 1390) is **DDS_BOOLEAN_FALSE** (p. 299), the returned collection is a 'set' containing each **DDSDataReader** (p. 1087) at most once, in no specified order.

If the application is within a **begin_access()** (p. 1405)/end_access() block, and the **PRESENTATION** (p. 351) policy of the **DDSSubscriber** (p. 1390) is **DDS_GROUP_PRESENTATION_QOS** (p. 352) or **DDS_HIGHEST_OFFERED_PRESENTATION_QOS** (p. 352), and **DDS_PresentationQosPolicy::ordered_access** (p. 827) in the **DDSSubscriber** (p. 1390) is **DDS_BOOLEAN_TRUE** (p. 298), the returned collection is a 'list' of DataReaders where a DataReader may appear more than one time.

To retrieve the samples in the order they were published across DataWriters of the same group (**DDSPublisher** (p. 1346) configured with **DDS_GROUP_PRESENTATION_QOS** (p. 352)), the application should read()/take() from each DataReader in the same order as it appears in the output sequence. The application will move to the next DataReader when the read()/take() operation fails with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

readers *<<inout>> (p. 200)* a DDS_DataReaderSeq object where the set or list of readers will be returned.

sample_states *<<in>> (p. 200)* the returned DataReader must contain samples that have one of these `sample_states`.

view_states *<<in>> (p. 200)* the returned DataReader must contain samples that have one of these `view_states`.

instance_states *<<in>> (p. 200)* the returned DataReader must contain samples that have one of these `instance_states`.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_**

NOT_ENABLED (p. 315).

See also:

Access to data samples (p. 95)
DDSSubscriber::begin_access (p. 1405)
DDSSubscriber::end_access (p. 1406)
PRESENTATION (p. 351)

6.218.2.17 virtual DDS_ReturnCode_t DDSSubscriber::get_all_datareaders (DDSDataReaderSeq & *readers*) [pure virtual]

Retrieve all the DataReaders created from this Subscriber.

Parameters:

readers <<***inout***>> *(p. 200)* Sequence where the DataReaders will be added

Returns:

One of the **Standard Return Codes** (p. 314)

6.218.2.18 virtual DDS_ReturnCode_t DDSSubscriber::notify_datareaders () [pure virtual]

Invokes the operation **DDSDataReaderListener::on_data_available()** (p. 1110) on the **DDSDataReaderListener** (p. 1108) objects attached to contained **DDSDataReader** (p. 1087) entities with **DDS_DATA_AVAILABLE_STATUS** (p. 324) that is considered changed as described in **Changes in read communication status** (p. 320).

This operation is typically invoked from the **DDSSubscriberListener::on_data_on_readers** (p. 1415) operation in the **DDSSubscriberListener** (p. 1414). That way the **DDSSubscriberListener** (p. 1414) can delegate to the **DDSDataReaderListener** (p. 1108) objects the handling of the data.

The operation will notify the data readers that have a `sample_state` of **DDS_NOT_READ_SAMPLE_STATE** (p. 112), `view_state` of **DDS_ANY_SAMPLE_STATE** (p. 112) and `instance_state` of **DDS_ANY_INSTANCE_STATE** (p. 117).

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.218.2.19 virtual DDSDomainParticipant* DDSSubscriber::get_participant () [pure virtual]

Returns the **DDSDomainParticipant** (p. 1139) to which the **DDSSubscriber** (p. 1390) belongs.

Returns:

the **DDSDomainParticipant** (p. 1139) to which the **DDSSubscriber** (p. 1390) belongs.

6.218.2.20 virtual DDS_ReturnCode_t DDSSubscriber::copy_from_topic_qos (DDS_DataReaderQos & *datareader_qos*, const DDS_TopicQos & *topic_qos*) [pure virtual]

Copies the policies in the **DDS_TopicQos** (p. 965) to the corresponding policies in the **DDS_DataReaderQos** (p. 515).

Copies the policies in the **DDS_TopicQos** (p. 965) to the corresponding policies in the **DDS_DataReaderQos** (p. 515) (replacing values in the **DDS_DataReaderQos** (p. 515), if present).

This is a "convenience" operation most useful in combination with the operations **DDSSubscriber::get_default_datareader_qos** (p. 1394) and **DDSTopic::get_qos** (p. 1423). The operation **DDSSubscriber::copy_from_topic_qos** (p. 1409) can be used to merge the **DDSDataReader** (p. 1087) default QoS policies with the corresponding ones on the **DDSTopic** (p. 1419). The resulting QoS can then be used to create a new **DDSDataReader** (p. 1087), or set its QoS.

This operation does not check the resulting **DDS_DataReaderQos** (p. 515) for consistency. This is because the 'merged' **DDS_DataReaderQos** (p. 515) may not be the final one, as the application can still modify some policies prior to applying the policies to the **DDSDataReader** (p. 1087).

Parameters:

datareader_qos *<<inout>> (p. 200)* **DDS_DataReaderQos** (p. 515) to be filled-up.

topic_qos *<<in>> (p. 200)* **DDS_TopicQos** (p. 965) to be merged with **DDS_DataReaderQos** (p. 515).

Returns:

One of the **Standard Return Codes** (p. 314)

6.218.2.21 virtual DDS_ReturnCode_t DDSSubscriber::set_qos (const DDS_SubscriberQos & *qos*) [pure virtual]

Sets the subscriber QoS.

This operation modifies the QoS of the **DDSSubscriber** (p. 1390).

The **DDS_SubscriberQos::group_data** (p. 935), **DDS_SubscriberQos::partition** (p. 935) and **DDS_SubscriberQos::entity_factory** (p. 935) can be changed. The other policies are immutable.

Parameters:

qos <<*in*>> *(p. 200)* **DDS_SubscriberQos** (p. 934) to be set to. Policies must be consistent. Immutable policies cannot be changed after **DDSSubscriber** (p. 1390) is enabled. The special value **DDS_SUBSCRIBER_QOS_DEFAULT** (p. 39) can be used to indicate that the QoS of the **DDSSubscriber** (p. 1390) should be changed to match the current default **DDS_SubscriberQos** (p. 934) set in the **DDSDomainParticipant** (p. 1139).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_SubscriberQos (p. 934) for rules on consistency among QoS
set_qos (abstract) (p. 1254)
Operations Allowed in Listener Callbacks (p. 1320)

6.218.2.22 virtual DDS_ReturnCode_t DDSSubscriber::set_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<*eXtension*>> *(p. 199)* Change the QoS of this subscriber using the input XML QoS profile.

This operation modifies the QoS of the **DDSSubscriber** (p. 1390).

The **DDS_SubscriberQos::group_data** (p. 935), **DDS_SubscriberQos::partition** (p. 935) and **DDS_SubscriberQos::entity_factory** (p. 935) can be changed. The other policies are immutable.

Parameters:

library_name <<*in*>> *(p. 200)* Library name containing the XML QoS

profile. If library_name is null RTI Connext will use the default library (see **DDSSubscriber::set_default_library** (p. 1397)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSSubscriber::set_default_profile** (p. 1398)).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315), or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315).

See also:

DDS_SubscriberQos (p. 934) for rules on consistency among QoS
Operations Allowed in Listener Callbacks (p. 1320)

6.218.2.23 virtual DDS_ReturnCode_t DDSSubscriber::get_qos (DDS_SubscriberQos & *qos*) [pure virtual]

Gets the subscriber QoS.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos *<<in>> (p. 200)* **DDS_SubscriberQos** (p. 934) to be filled in.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

get_qos (abstract) (p. 1255)

6.218.2.24 virtual DDS_ReturnCode_t DDSSubscriber::set_listener (DDSSubscriberListener * *l*, DDS_StatusMask *mask* = DDS_STATUS_MASK_ALL) [pure virtual]

Sets the subscriber listener.

Parameters:

l *<<in>> (p. 200)* **DDSSubscriberListener** (p. 1414) to set to.

mask <<*in*>> *(p. 200)* **DDS_StatusMask** (p. 321) associated with the **DDSSubscriberListener** (p. 1414).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

set_listener (abstract) (p. 1255)

6.218.2.25 virtual DDSSubscriberListener* DDSSubscriber::get_listener () [pure virtual]

Get the subscriber listener.

Returns:

DDSSubscriberListener (p. 1414) of the **DDSSubscriber** (p. 1390).

See also:

get_listener (abstract) (p. 1256)

6.218.2.26 virtual DDSDataReader* DDSSubscriber::lookup_datareader_by_name_exp (const char * *datareader_name*) [pure virtual]

<<*experimental*>> *(p. 199)* <<*eXtension*>> *(p. 199)* Retrieves a **DDSDataReader** (p. 1087) contained within the **DDSSubscriber** (p. 1390) the **DDSDataReader** (p. 1087) entity name.

Every **DDSDataReader** (p. 1087) in the system has an entity name which is configured and stored in the <<*eXtension*>> *(p. 199)* EntityName policy, **ENTITY_NAME** (p. 445).

This operation retrieves the **DDSDataReader** (p. 1087) within the **DDSSubscriber** (p. 1390) whose name matches the one specified. If there are several **DDSDataReader** (p. 1087) with the same name within the **DDSSubscriber** (p. 1390), the operation returns the first matching occurrence.

Parameters:

datareader_name <<*in*>> *(p. 200)* Entity name of the **DDSDataReader** (p. 1087).

Returns:

The first **DDSDataReader** (p. 1087) found with the specified name or NULL if it is not found.

See also:

DDSDomainParticipant::lookup_datareader_by_name_exp (p. 1214)

6.219 DDSSubscriberListener Class Reference

<<interface>> *(p. 199)* **DDSListener** (p. 1318) for status about a subscriber.

Inheritance diagram for DDSSubscriberListener::

Public Member Functions

- virtual void **on_data_on_readers** (**DDSSubscriber** *sub)

  Handles the ***DDS_DATA_ON_READERS_STATUS*** *(p. 324) communication status.*

6.219.1 Detailed Description

<<interface>> *(p. 199)* **DDSListener** (p. 1318) for status about a subscriber.

Entity:

DDSSubscriber (p. 1390)

Status:

DDS_DATA_AVAILABLE_STATUS (p. 324);
DDS_DATA_ON_READERS_STATUS (p. 324);
DDS_LIVELINESS_CHANGED_STATUS (p. 325), **DDS_LivelinessChangedStatus** (p. 775);
DDS_REQUESTED_DEADLINE_MISSED_STATUS (p. 323), **DDS_RequestedDeadlineMissedStatus** (p. 875);
DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS (p. 323), **DDS_RequestedIncompatibleQosStatus** (p. 877);
DDS_SAMPLE_LOST_STATUS (p. 324), **DDS_SampleLostStatus** (p. 923);
DDS_SAMPLE_REJECTED_STATUS (p. 324), **DDS_SampleRejectedStatus** (p. 925);
DDS_SUBSCRIPTION_MATCHED_STATUS (p. 326), **DDS_SubscriptionMatchedStatus** (p. 945)

See also:

DDSListener (p. 1318)
Status Kinds (p. 317)
Operations Allowed in Listener Callbacks (p. 1320)

6.219.2 Member Function Documentation

6.219.2.1 virtual void DDSSubscriberListener::on_data_on_readers (DDSSubscriber * *sub*) [virtual]

Handles the **DDS_DATA_ON_READERS_STATUS** (p. 324) communication status.

6.220 DDSSubscriberSeq Class Reference

Declares IDL `sequence` < **DDSSubscriber** (p. 1390) > .

6.220.1 Detailed Description

Declares IDL `sequence` < **DDSSubscriber** (p. 1390) > .

See also:

FooSeq (p. 1494)

6.221 DDSSubscriptionBuiltinTopicDataDataReader Class Reference

Instantiates `DataReader` < **DDS_SubscriptionBuiltinTopicData** (p. 936) > .

Inheritance diagram for DDSSubscriptionBuiltinTopicDataDataReader::

6.221.1 Detailed Description

Instantiates `DataReader` < **DDS_SubscriptionBuiltinTopicData** (p. 936) > .

DDSDataReader (p. 1087) of topic **DDS_SUBSCRIPTION_TOPIC_NAME** (p. 291) used for accessing **DDS_SubscriptionBuiltinTopicData** (p. 936) of the remote **DDSDataReader** (p. 1087) and the associated **DDSSubscriber** (p. 1390).

Instantiates:

<<*generic*>> *(p. 199)* **FooDataReader** (p. 1444)

See also:

DDS_SubscriptionBuiltinTopicData (p. 936)
DDS_SUBSCRIPTION_TOPIC_NAME (p. 291)

6.222 DDSSubscriptionBuiltinTopicDataTypeSupport Class Reference

Instantiates `TypeSupport` < **DDS_SubscriptionBuiltinTopicData** (p. 936) > .

6.222.1 Detailed Description

Instantiates `TypeSupport` < **DDS_SubscriptionBuiltinTopicData** (p. 936) > .

Instantiates:

<<*generic*>> *(p. 199)* **FooTypeSupport** (p. 1509)

See also:

DDS_SubscriptionBuiltinTopicData (p. 936)

6.223 DDSTopic Class Reference

<<***interface***>> *(*p. *199)* The most basic description of the data to be published and subscribed.

Inheritance diagram for DDSTopic::

Public Member Functions

- virtual **DDS_ReturnCode_t get_inconsistent_topic_status** (**DDS_InconsistentTopicStatus** &status)=0

  Allows the application to retrieve the ***DDS_INCONSISTENT_TOPIC_STATUS*** *(*p. *322) status of a* ***DDSTopic*** *(*p. *1419).*

- virtual **DDS_ReturnCode_t set_qos** (const **DDS_TopicQos** &qos)=0

  Set the topic QoS.

- virtual **DDS_ReturnCode_t set_qos_with_profile** (const char *library_name, const char *profile_name)=0

  <<**eXtension**>> (p. 199) *Change the QoS of this topic using the input XML QoS profile.*

- virtual **DDS_ReturnCode_t get_qos** (**DDS_TopicQos** &qos)=0

  Get the topic QoS.

- virtual **DDS_ReturnCode_t set_listener** (**DDSTopicListener** *l, **DDS_StatusMask** mask=DDS_STATUS_MASK_ALL)=0

  Set the topic listener.

- virtual **DDSTopicListener** * **get_listener** ()=0

  Get the topic listener.

Static Public Member Functions

- static **DDSTopic** * **narrow** (**DDSTopicDescription** *topic_description)

  Narrow the given ***DDSTopicDescription*** *(*p. *1427) pointer to a* ***DDSTopic*** *(*p. *1419) pointer.*

6.223.1 Detailed Description

*<<**interface**>>* *(p. 199)* The most basic description of the data to be published and subscribed.

QoS:

DDS_TopicQos (p. 965)

Status:

DDS_INCONSISTENT_TOPIC_STATUS (p. 322), **DDS_InconsistentTopicStatus** (p. 762)

Listener:

DDSTopicListener (p. 1430)

A **DDSTopic** (p. 1419) is identified by its name, which must be unique in the whole domain. In addition (by virtue of extending **DDSTopicDescription** (p. 1427)) it fully specifies the type of the data that can be communicated when publishing or subscribing to the **DDSTopic** (p. 1419).

DDSTopic (p. 1419) is the only **DDSTopicDescription** (p. 1427) that can be used for publications and therefore associated with a **DDSDataWriter** (p. 1113).

The following operations may be called even if the **DDSTopic** (p. 1419) is not enabled. Other operations will fail with the value **DDS_RETCODE_NOT_ENABLED** (p. 315) if called on a disabled **DDSTopic** (p. 1419):

- All the base-class operations **set_qos()** (p. 1421), **set_qos_with_profile()** (p. 1422), **get_qos()** (p. 1423), **set_listener()** (p. 1423), **get_listener()** (p. 1424) **enable()** (p. 1256), **get_statuscondition()** (p. 1257) and **get_status_changes()** (p. 1257)

- **get_inconsistent_topic_status()** (p. 1421)

See also:

Operations Allowed in Listener Callbacks (p. 1320)

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

6.223.2 Member Function Documentation

6.223.2.1 static DDSTopic* DDSTopic::narrow (DDSTopicDescription * *topic_description*) [static]

Narrow the given **DDSTopicDescription** (p. 1427) pointer to a **DDSTopic** (p. 1419) pointer.

Returns:

DDSTopic (p. 1419) if this **DDSTopicDescription** (p. 1427) is a **DDSTopic** (p. 1419). Otherwise, return NULL.

6.223.2.2 virtual DDS_ReturnCode_t DDSTopic::get_inconsistent_topic_status (DDS_InconsistentTopicStatus & *status*) [pure virtual]

Allows the application to retrieve the **DDS_INCONSISTENT_TOPIC_STATUS** (p. 322) status of a **DDSTopic** (p. 1419).

Retrieve the current **DDS_InconsistentTopicStatus** (p. 762)

Parameters:

status *<<**inout**>> (p. 200)* Status to be retrieved.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

DDS_InconsistentTopicStatus (p. 762)

6.223.2.3 virtual DDS_ReturnCode_t DDSTopic::set_qos (const DDS_TopicQos & *qos*) [pure virtual]

Set the topic QoS.

The **DDS_TopicQos::topic_data** (p. 966) and **DDS_TopicQos::deadline** (p. 967), **DDS_TopicQos::latency_budget** (p. 967), **DDS_TopicQos::transport_priority** (p. 967) and **DDS_TopicQos::lifespan** (p. 968) can be changed. The other policies are immutable.

Parameters:

qos *<<**in**>> (p. 200)* Set of policies to be applied to **DDSTopic** (p. 1419).

Policies must be consistent. Immutable policies cannot be changed after **DDSTopic** (p. 1419) is enabled. The special value **DDS_TOPIC_QOS_DEFAULT** (p. 38) can be used to indicate that the QoS of the **DDSTopic** (p. 1419) should be changed to match the current default **DDS_TopicQos** (p. 965) set in the **DDSDomainParticipant** (p. 1139).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) if immutable policy is changed, or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) if policies are inconsistent

See also:

DDS_TopicQos (p. 965) for rules on consistency among QoS
set_qos (abstract) (p. 1254)
Operations Allowed in Listener Callbacks (p. 1320)

6.223.2.4 virtual DDS_ReturnCode_t DDSTopic::set_qos_with_profile (const char * *library_name*, const char * *profile_name*) [pure virtual]

<<eXtension>> (p. 199) Change the QoS of this topic using the input XML QoS profile.

The **DDS_TopicQos::topic_data** (p. 966) and **DDS_TopicQos::deadline** (p. 967), **DDS_TopicQos::latency_budget** (p. 967), **DDS_TopicQos::transport_priority** (p. 967) and **DDS_TopicQos::lifespan** (p. 968) can be changed. The other policies are immutable.

Parameters:

library_name *<<in>> (p. 200)* Library name containing the XML QoS profile. If library_name is null RTI Connext will use the default library (see **DDSDomainParticipant::set_default_library** (p. 1159)).

profile_name *<<in>> (p. 200)* XML QoS Profile name. If profile_name is null RTI Connext will use the default profile (see **DDSDomainParticipant::set_default_profile** (p. 1160)).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_IMMUTABLE_POLICY** (p. 315) if immutable policy is changed, or **DDS_RETCODE_INCONSISTENT_POLICY** (p. 315) if policies are inconsistent

See also:

DDS_TopicQos (p. 965) for rules on consistency among QoS
Operations Allowed in Listener Callbacks (p. 1320)

6.223.2.5 virtual DDS_ReturnCode_t DDSTopic::get_qos (DDS_TopicQos & *qos*) [pure virtual]

Get the topic QoS.

This method may potentially allocate memory depending on the sequences contained in some QoS policies.

Parameters:

qos *<<**inout**>>* *(p. 200)* QoS to be filled up.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

get_qos (abstract) (p. 1255)

6.223.2.6 virtual DDS_ReturnCode_t DDSTopic::set_listener (DDSTopicListener * *l*, DDS_StatusMask *mask* = DDS_STATUS_MASK_ALL) [pure virtual]

Set the topic listener.

Parameters:

l *<<**in**>>* *(p. 200)* Listener to be installed on entity.

mask *<<**in**>>* *(p. 200)* Changes of communication status to be invoked on the listener. See **DDS_StatusMask** (p. 321).

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

set_listener (abstract) (p. 1255)

6.223.2.7 virtual DDSTopicListener* DDSTopic::get_listener () [pure virtual]

Get the topic listener.

Returns:

Existing listener attached to the **DDSTopic** (p. 1419).

See also:

get_listener (abstract) (p. 1256)

6.224 DDSTopicBuiltinTopicDataDataReader Class Reference

Instantiates `DataReader` < **DDS_TopicBuiltinTopicData** (p. 958) > .

Inheritance diagram for DDSTopicBuiltinTopicDataDataReader::

6.224.1 Detailed Description

Instantiates `DataReader` < **DDS_TopicBuiltinTopicData** (p. 958) > .

DDSDataReader (p. 1087) of topic **DDS_TOPIC_TOPIC_NAME** (p. 287) used for accessing **DDS_TopicBuiltinTopicData** (p. 958) of the remote **DDSTopic** (p. 1419).

Note: The **DDS_TopicBuiltinTopicData** (p. 958) built-in topic is meant to convey information about discovered Topics. This Topic's samples are not propagated in a separate packet on the wire. Instead, the data is sent as part of the information carried by other built-in topics (**DDS_PublicationBuiltinTopicData** (p. 839) and **DDS_SubscriptionBuiltinTopicData** (p. 936)). Therefore TopicBuiltinTopicData DataReaders will not receive any data.

Instantiates:

<<*generic*>> *(p. 199)* **FooDataReader** (p. 1444)

See also:

DDS_TopicBuiltinTopicData (p. 958)
DDS_TOPIC_TOPIC_NAME (p. 287)

6.225 DDSTopicBuiltinTopicDataTypeSupport Class Reference

Instantiates `TypeSupport` < **DDS_TopicBuiltinTopicData** (p. 958) > .

6.225.1 Detailed Description

Instantiates `TypeSupport` < **DDS_TopicBuiltinTopicData** (p. 958) > .

Instantiates:

<<*generic*>> *(p. 199)* **FooTypeSupport** (p. 1509)

See also:

DDS_TopicBuiltinTopicData (p. 958)

6.226 DDSTopicDescription Class Reference

<<*interface*>> *(p. 199)* Base class for **DDSTopic** (p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322).

Inheritance diagram for DDSTopicDescription::

Public Member Functions

- virtual const char * **get_type_name** ()=0

  Get the associated `type_name`.

- virtual const char * **get_name** ()=0

  Get the name used to create this ***DDSTopicDescription*** *(p. 1427) .*

- virtual **DDSDomainParticipant** * **get_participant** ()=0

  Get the ***DDSDomainParticipant*** *(p. 1139) to which the* ***DDSTopicDescription*** *(p. 1427) belongs.*

6.226.1 Detailed Description

<<*interface*>> *(p. 199)* Base class for **DDSTopic** (p. 1419), **DDSContentFilteredTopic** (p. 1081), and **DDSMultiTopic** (p. 1322).

DDSTopicDescription (p. 1427) represents the fact that both publications and subscriptions are tied to a single data-type. Its attribute `type_name` defines a unique resulting type for the publication or the subscription and therefore creates an implicit association with a **DDSTypeSupport** (p. 1432).

DDSTopicDescription (p. 1427) has also a `name` that allows it to be retrieved locally.

See also:

DDSTypeSupport (p. 1432), **FooTypeSupport** (p. 1509)

6.226.2 Member Function Documentation

6.226.2.1 virtual const char* DDSTopicDescription::get_type_name () [pure virtual]

Get the associated `type_name`.

The type name defines a locally unique type for the publication or the subscription.

The `type_name` corresponds to a unique string used to register a type via the **FooTypeSupport::register_type** (p. 1510) method.

Thus, the `type_name` implies an association with a corresponding **DDSTypeSupport** (p. 1432) and this **DDSTopicDescription** (p. 1427).

Returns:

the type name. The returned type name is valid until the **DDSTopicDescription** (p. 1427) is deleted.

Postcondition:

The result is non-NULL.

See also:

DDSTypeSupport (p. 1432), **FooTypeSupport** (p. 1509)

6.226.2.2 virtual const char* DDSTopicDescription::get_name () [pure virtual]

Get the name used to create this **DDSTopicDescription** (p. 1427) .

Returns:

the name used to create this **DDSTopicDescription** (p. 1427). The returned topic name is valid until the **DDSTopicDescription** (p. 1427) is deleted.

Postcondition:

The result is non-NULL.

6.226.2.3 virtual DDSDomainParticipant* DDSTopicDescription::get_participant () [pure virtual]

Get the **DDSDomainParticipant** (p. 1139) to which the **DDSTopicDescription** (p. 1427) belongs.

Returns:

The **DDSDomainParticipant** (p. 1139) to which the **DDSTopicDescription** (p. 1427) belongs.

Postcondition:

The result is non-NULL.

6.227 DDSTopicListener Class Reference

<<*interface*>> *(*p. *199)* **DDSListener** (p. 1318) for **DDSTopic** (p. 1419) entities.

Inheritance diagram for DDSTopicListener::

Public Member Functions

- virtual void **on_inconsistent_topic** (**DDSTopic** *topic, const **DDS_InconsistentTopicStatus** &status)

  *Handle the **DDS_INCONSISTENT_TOPIC_STATUS** (*p. *322) status.*

6.227.1 Detailed Description

<<*interface*>> *(*p. *199)* **DDSListener** (p. 1318) for **DDSTopic** (p. 1419) entities.

Entity:

DDSTopic (p. 1419)

Status:

DDS_INCONSISTENT_TOPIC_STATUS (p. 322), **DDS_InconsistentTopicStatus** (p. 762)

This is the interface that can be implemented by an application-provided class and then registered with the **DDSTopic** (p. 1419) such that the application can be notified by RTI Connext of relevant status changes.

See also:

Status Kinds (p. 317)
DDSListener (p. 1318)
DDSTopic::set_listener (p. 1423)
Operations Allowed in Listener Callbacks (p. 1320)

6.227.2 Member Function Documentation

6.227.2.1 virtual void DDSTopicListener::on_inconsistent_topic (DDSTopic * *topic*, const DDS_InconsistentTopicStatus & *status*) [virtual]

Handle the **DDS_INCONSISTENT_TOPIC_STATUS** (p. 322) status.

This callback is called when a remote **DDSTopic** (p. 1419) is discovered but is inconsistent with the locally created **DDSTopic** (p. 1419) of the same topic name.

Parameters:

topic <<*out*>> *(p. 200)* Locally created **DDSTopic** (p. 1419) that triggers the listener callback

status <<*out*>> *(p. 200)* Current inconsistent status of locally created **DDSTopic** (p. 1419)

6.228 DDSTypeSupport Class Reference

<<*interface*>> *(*p. *199)* An abstract *marker* interface that has to be specialized for each concrete user data type that will be used by the application.

Inheritance diagram for DDSTypeSupport::

6.228.1 Detailed Description

<<*interface*>> *(*p. *199)* An abstract *marker* interface that has to be specialized for each concrete user data type that will be used by the application.

The implementation provides an automatic means to generate a type-specific class, **FooTypeSupport** (p. 1509), from a description of the type in IDL.

A **DDSTypeSupport** (p. 1432) must be registered using the **FooTypeSupport::register_type** (p. 1510) operation on this type-specific class before it can be used to create **DDSTopic** (p. 1419) objects.

See also:

FooTypeSupport (p. 1509)
rtiddsgen (p. 220)

Examples:

HelloWorldSupport.cxx.

6.229 DDSWaitSet Class Reference

<<*interface*>> *(p. 199)* Allows an application to wait until one or more of the attached **DDSCondition** (p. 1075) objects has a `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298) or else until the timeout expires.

Public Member Functions

- virtual **DDS_ReturnCode_t wait** (**DDSConditionSeq** &active_conditions, const **DDS_Duration_t** &timeout)

  Allows an application thread to wait for the occurrence of certain conditions.

- virtual **DDS_ReturnCode_t attach_condition** (**DDSCondition** *cond)

  *Attaches a **DDSCondition** (p. 1075) to the **DDSWaitSet** (p. 1433).*

- virtual **DDS_ReturnCode_t detach_condition** (**DDSCondition** *cond)

  *Detaches a **DDSCondition** (p. 1075) from the **DDSWaitSet** (p. 1433).*

- virtual **DDS_ReturnCode_t get_conditions** (**DDSConditionSeq** &attached_conditions)

  *Retrieves the list of attached **DDSCondition** (p. 1075) (s).*

- virtual **DDS_ReturnCode_t set_property** (const **DDS_WaitSetProperty_t** &prop)

  <<**eXtension**>> (p. 199) *Sets the **DDS_WaitSetProperty_t** (p. 1056), to configure the associated **DDSWaitSet** (p. 1433) to return after one or more trigger events have occurred.*

- virtual **DDS_ReturnCode_t get_property** (**DDS_WaitSetProperty_t** &prop)

  <<**eXtension**>> (p. 199) *Retrieves the **DDS_WaitSetProperty_t** (p. 1056) configuration of the associated **DDSWaitSet** (p. 1433).*

- virtual ~**DDSWaitSet** ()

  Destructor.

- **DDSWaitSet** ()

  Default no-argument constructor.

- **DDSWaitSet** (const **DDS_WaitSetProperty_t** &prop)

<<**eXtension**>> (p. 199) *Constructor for a* ***DDSWaitSet*** *(p. 1433) that may delay for more while specifying that will be woken up after the given number of events or delay period, whichever happens first*

6.229.1 Detailed Description

<<***interface***>> *(p. 199)* Allows an application to wait until one or more of the attached **DDSCondition** (p. 1075) objects has a `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298) or else until the timeout expires.

6.229.2 Usage

DDSCondition (p. 1075) (s) (in conjunction with wait-sets) provide an alternative mechanism to allow the middleware to communicate communication status changes (including arrival of data) to the application.

This mechanism is wait-based. Its general use pattern is as follows:

- The application indicates which relevant information it wants to get by creating **DDSCondition** (p. 1075) objects (**DDSStatusCondition** (p. 1376), **DDSReadCondition** (p. 1374) or **DDSQueryCondition** (p. 1372)) and attaching them to a **DDSWaitSet** (p. 1433).

- It then waits on that **DDSWaitSet** (p. 1433) until the `trigger_value` of one or several **DDSCondition** (p. 1075) objects become **DDS_BOOLEAN_TRUE** (p. 298).

- It then uses the result of the wait (i.e., `active_conditions`, the list of **DDSCondition** (p. 1075) objects with `trigger_value` == **DDS_BOOLEAN_TRUE** (p. 298)) to actually get the information:
  - by calling **DDSEntity::get_status_changes** (p. 1257) and then `get_<communication_status>()` on the relevant **DDSEntity** (p. 1253), if the condition is a **DDSStatusCondition** (p. 1376) and the status changes, refer to plain communication status;
  - by calling **DDSEntity::get_status_changes** (p. 1257) and then **DDSSubscriber::get_datareaders** (p. 1407) on the relevant **DDSSubscriber** (p. 1390) (and then **FooDataReader::read()** (p. 1447) or **FooDataReader::take** (p. 1448) on the returned **DDSDataReader** (p. 1087) objects), if the condition is a **DDSStatusCondition** (p. 1376) and the status changes refers to **DDS_DATA_ON_READERS_STATUS** (p. 324);

- by calling **DDSEntity::get_status_changes** (p. 1257) and then **FooDataReader::read()** (p. 1447) or **FooDataReader::take** (p. 1448) on the relevant **DDSDataReader** (p. 1087), if the condition is a **DDSStatusCondition** (p. 1376) and the status changes refers to **DDS_DATA_AVAILABLE_STATUS** (p. 324);
- by calling directly **FooDataReader::read_w_condition** (p. 1454) or **FooDataReader::take_w_condition** (p. 1456) on a **DDSDataReader** (p. 1087) with the **DDSCondition** (p. 1075) as a parameter if it is a **DDSReadCondition** (p. 1374) or a **DDSQueryCondition** (p. 1372).

Usually the first step is done in an initialization phase, while the others are put in the application main loop.

As there is no extra information passed from the middleware to the application when a wait returns (only the list of triggered **DDSCondition** (p. 1075) objects), **DDSCondition** (p. 1075) objects are meant to embed all that is needed to react properly when enabled. In particular, **DDSEntity** (p. 1253)-related conditions are related to exactly one **DDSEntity** (p. 1253) and cannot be shared.

The blocking behavior of the **DDSWaitSet** (p. 1433) is illustrated below.

The result of a **DDSWaitSet::wait** (p. 1438) operation depends on the state of the **DDSWaitSet** (p. 1433), which in turn depends on whether at least one attached **DDSCondition** (p. 1075) has a `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298). If the wait operation is called on **DDSWaitSet** (p. 1433) with state BLOCKED, it will block the calling thread. If wait is called on a **DDSWaitSet** (p. 1433) with state UNBLOCKED, it will return immediately. In addition, when the **DDSWaitSet** (p. 1433) transitions from BLOCKED to UNBLOCKED it wakes up any threads that had called wait on it.

A key aspect of the **DDSCondition** (p. 1075)/**DDSWaitSet** (p. 1433) mechanism is the setting of the `trigger_value` of each **DDSCondition** (p. 1075).

6.229.3 Trigger State of a ::DDSStatusCondition

The `trigger_value` of a **DDSStatusCondition** (p. 1376) is the boolean OR of the ChangedStatusFlag of all the communication statuses (see **Status Kinds** (p. 317)) to which it is sensitive. That is, `trigger_value` == **DDS_BOOLEAN_FALSE** (p. 299) only if all the values of the ChangedStatusFlags are **DDS_BOOLEAN_FALSE** (p. 299).

The sensitivity of the **DDSStatusCondition** (p. 1376) to a particular communication status is controlled by the list of `enabled_statuses` set on the condition by means of the **DDSStatusCondition::set_enabled_statuses** (p. 1377) operation.

6.229.4 Trigger State of a ::DDSReadCondition

Similar to the **DDSStatusCondition** (p. 1376), a **DDSReadCondition** (p. 1374) also has a `trigger_value` that determines whether the attached **DDSWaitSet** (p. 1433) is BLOCKED or UNBLOCKED. However, unlike the **DDSStatusCondition** (p. 1376), the `trigger_value` of the **DDSReadCondition** (p. 1374) is tied to the presence of *at least a sample* managed by RTI Connext with **DDS_SampleStateKind** (p. 111) and **DDS_ViewStateKind** (p. 113) matching those of the **DDSReadCondition** (p. 1374). Furthermore, for the **DDSQueryCondition** (p. 1372) to have a `trigger_value` == **DDS_BOOLEAN_TRUE** (p. 298), the data associated with the sample must be such that the `query_expression` evaluates to **DDS_BOOLEAN_TRUE** (p. 298).

The fact that the trigger_value of a **DDSReadCondition** (p. 1374) depends on the presence of samples on the associated **DDSDataReader** (p. 1087) implies that a single take operation can potentially change the `trigger_value` of several **DDSReadCondition** (p. 1374) or **DDSQueryCondition** (p. 1372) conditions. For example, if all samples are taken, any **DDSReadCondition** (p. 1374) and **DDSQueryCondition** (p. 1372) conditions associated with the **DDSDataReader** (p. 1087) that had their `trigger_value==TRUE` before will see the `trigger_value` change to FALSE. Note that this does not guarantee that **DDSWaitSet** (p. 1433) objects that were separately attached to those conditions will not be woken up. Once we have `trigger_value==TRUE` on a condition, it may wake up the attached **DDSWaitSet** (p. 1433), the condition transitioning to `trigger_value==FALSE` does not necessarily 'unwakeup' the WaitSet as 'unwakening' may not be possible in general.

The consequence is that an application blocked on a **DDSWaitSet** (p. 1433) may return from the wait with a list of conditions, some of which are not no longer 'active'. This is unavoidable if multiple threads are concurrently waiting on separate **DDSWaitSet** (p. 1433) objects and taking data associated with the same **DDSDataReader** (p. 1087) entity.

To elaborate further, consider the following example: A **DDSReadCondition** (p. 1374) that has a `sample_state_mask` = {**DDS_NOT_READ_SAMPLE_STATE** (p. 112)} will have `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298) whenever a new sample arrives and will transition to **DDS_BOOLEAN_FALSE** (p. 299) as soon as all the newly-arrived samples are either read (so their sample state changes to READ) or taken (so they are no longer managed by RTI Connext). However if the same **DDSReadCondition** (p. 1374) had a `sample_state_mask` = { **DDS_READ_SAMPLE_STATE** (p. 112), **DDS_NOT_READ_SAMPLE_STATE** (p. 112) }, then the `trigger_value` would only become **DDS_BOOLEAN_FALSE** (p. 299) once all the newly-arrived samples are taken (it is not sufficient to read them as that would only change the sample state to READ), which overlaps the mask on the **DDSReadCondition** (p. 1374).

6.229.5 Trigger State of a ::DDSGuardCondition

The `trigger_value` of a **DDSGuardCondition** (p. 1263) is completely controlled by the application via the operation **DDSGuardCondition::set_trigger_value** (p. 1264).

See also:

Status Kinds (p. 317)
DDSStatusCondition (p. 1376), **DDSGuardCondition** (p. 1263)
DDSListener (p. 1318)

6.229.6 Constructor & Destructor Documentation

6.229.6.1 virtual DDSWaitSet::~DDSWaitSet () [virtual]

Destructor.

Releases the resources asociated with this **DDSWaitSet** (p. 1433).

Freeing a null pointer is safe and does nothing.

6.229.6.2 DDSWaitSet::DDSWaitSet ()

Default no-argument constructor.

Construct a new **DDSWaitSet** (p. 1433).

Returns:

A new **DDSWaitSet** (p. 1433) or NULL if one could not be allocated.

6.229.6.3 DDSWaitSet::DDSWaitSet (const DDS_WaitSetProperty_t & *prop*)

<<***eXtension***>> *(p. 199)* Constructor for a **DDSWaitSet** (p. 1433) that may delay for more while specifying that will be woken up after the given number of events or delay period, whichever happens first

Constructs a new **DDSWaitSet** (p. 1433).

Returns:

A new **DDSWaitSet** (p. 1433) or NULL if one could not be allocated.

6.229.7 Member Function Documentation

6.229.7.1 virtual DDS_ReturnCode_t DDSWaitSet::wait (DDSConditionSeq & *active_conditions*, const DDS_Duration_t & *timeout*) [virtual]

Allows an application thread to wait for the occurrence of certain conditions.

If none of the conditions attached to the **DDSWaitSet** (p. 1433) have a `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298), the wait operation will block suspending the calling thread.

The result of the wait operation is the list of all the attached conditions that have a `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298) (i.e., the conditions that unblocked the wait).

Note: The resolution of the `timeout` period is constrained by the resolution of the system clock.

The wait operation takes a `timeout` argument that specifies the maximum duration for the wait. If this duration is exceeded and none of the attached **DDSCondition** (p. 1075) objects is **DDS_BOOLEAN_TRUE** (p. 298), wait will return with the return code **DDS_RETCODE_TIMEOUT** (p. 316). In this case, the resulting list of conditions will be empty.

Note: The resolution of the `timeout` period is constrained by the resolution of the system clock.

DDS_RETCODE_TIMEOUT (p. 316) will *not* be returned when the `timeout` duration is exceeded if attached **DDSCondition** (p. 1075) objects are **DDS_BOOLEAN_TRUE** (p. 298), or in the case of a **DDSWaitSet** (p. 1433) waiting for more than one trigger event, if one or more trigger events have occurred.

It is not allowable for for more than one application thread to be waiting on the same **DDSWaitSet** (p. 1433). If the wait operation is invoked on a **DDSWaitSet** (p. 1433) that already has a thread blocking on it, the operation will return immediately with the value **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

Parameters:

active_conditions *<<inout>> (p. 200)* a valid non-NULL **DDSConditionSeq** (p. 1076) object. Note that RTI Connext will not allocate a new object if `active_conditions` is NULL; the method will return **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

timeout *<<in>> (p. 200)* a wait timeout

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_-PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_-TIMEOUT** (p. 316).

6.229.7.2 virtual DDS_ReturnCode_t DDSWaitSet::attach_-condition (DDSCondition * *cond*)

`[virtual]`

Attaches a **DDSCondition** (p. 1075) to the **DDSWaitSet** (p. 1433).

It is possible to attach a **DDSCondition** (p. 1075) on a **DDSWaitSet** (p. 1433) that is currently being waited upon (via the wait operation). In this case, if the **DDSCondition** (p. 1075) has a `trigger_value` of **DDS_BOOLEAN_TRUE** (p. 298), then attaching the condition will unblock the **DDSWaitSet** (p. 1433).

Parameters:

cond *<<in>>* *(p. 200)* Contition to be attached.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_-OUT_OF_RESOURCES** (p. 315).

6.229.7.3 virtual DDS_ReturnCode_t DDSWaitSet::detach_-condition (DDSCondition * *cond*)

`[virtual]`

Detaches a **DDSCondition** (p. 1075) from the **DDSWaitSet** (p. 1433).

If the **DDSCondition** (p. 1075) was not attached to the **DDSWaitSet** (p. 1433) the operation will return **DDS_RETCODE_BAD_PARAMETER** (p. 315).

Parameters:

cond *<<in>>* *(p. 200)* Condition to be detached.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_-PRECONDITION_NOT_MET** (p. 315).

6.229.7.4 virtual DDS_ReturnCode_t DDSWaitSet::get_conditions (DDSConditionSeq & *attached_conditions*) [virtual]

Retrieves the list of attached **DDSCondition** (p. 1075) (s).

Parameters:

attached_conditions *<<inout>>* *(p. 200)* a **DDSConditionSeq** (p. 1076) object where the list of attached conditions will be returned

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

6.229.7.5 virtual DDS_ReturnCode_t DDSWaitSet::set_property (const DDS_WaitSetProperty_t & *prop*) [virtual]

<<eXtension>> *(p. 199)* Sets the **DDS_WaitSetProperty_t** (p. 1056), to configure the associated **DDSWaitSet** (p. 1433) to return after one or more trigger events have occurred.

Parameters:

prop *<<in>>* *(p. 200)*

Returns:

One of the **Standard Return Codes** (p. 314)

6.229.7.6 virtual DDS_ReturnCode_t DDSWaitSet::get_property (DDS_WaitSetProperty_t & *prop*) [virtual]

<<eXtension>> *(p. 199)* Retrieves the **DDS_WaitSetProperty_t** (p. 1056) configuration of the associated **DDSWaitSet** (p. 1433).

Parameters:

prop *<<out>>* *(p. 200)*

Returns:

One of the **Standard Return Codes** (p. 314)

6.230 DDSWriterContentFilter Class Reference

Inheritance diagram for DDSWriterContentFilter::

Public Member Functions

- virtual **DDS_ReturnCode_t writer_compile** (void *writer_filter_data, **DDS_ExpressionProperty** *prop, const char *expression, const **DDS_StringSeq** *parameters, const **DDS_TypeCode** *type_code, const char *type_class_name, const **DDS_Cookie_t** *cookie)=0

- virtual struct **DDS_CookieSeq** * **writer_evaluate** (void *writer_filter_data, const void *sample, const **DDS_FilterSampleInfo** *meta_data)=0

- virtual void **writer_finalize** (void *writer_filter_data, const **DDS_Cookie_t** *cookie)=0

- virtual **DDS_ReturnCode_t writer_attach** (void **writer_filter_data)=0

- virtual void **writer_detach** (void *writer_filter_data)=0

- virtual void **writer_return_loan** (void *writer_filter_data, **DDS_CookieSeq** *cookies)=0

6.230.1 Detailed Description

6.230.2 Member Function Documentation

6.230.2.1 virtual DDS_ReturnCode_t DDSWriterContentFilter::writer_compile (void * *writer_filter_data*, DDS_ExpressionProperty * *prop*, const char * *expression*, const DDS_StringSeq * *parameters*, const DDS_TypeCode * *type_code*, const char * *type_class_name*, const DDS_Cookie_t * *cookie*) [pure virtual]

6.230.2.2 virtual struct DDS_CookieSeq* DDSWriterContentFilter::writer_evaluate (void * *writer_filter_data*, const void * *sample*, const DDS_FilterSampleInfo * *meta_data*) [read, pure virtual]

6.230.2.3 virtual void DDSWriterContentFilter::writer_finalize (void * *writer_filter_data*, const DDS_Cookie_t * *cookie*) [pure virtual]

6.230.2.4 virtual DDS_ReturnCode_t DDSWriterContentFilter::writer_attach (void ** *writer_filter_data*) [pure virtual]

6.230.2.5 virtual void DDSWriterContentFilter::writer_detach (void * *writer_filter_data*) [pure virtual]

6.230.2.6 virtual void DDSWriterContentFilter::writer_return_loan (void * *writer_filter_data*, DDS_CookieSeq * *cookies*) [pure virtual]

6.231 Foo Struct Reference

A representative user-defined data type.

6.231.1 Detailed Description

A representative user-defined data type.

Foo (p. 1443) represents a user-defined data-type that is intended to be distributed using DDS.

The type **Foo** (p. 1443) is usually defined using IDL syntax and placed in a ".idl" file that is then processed using **rtiddsgen** (p. 220). The **rtiddsgen** (p. 220) utility generates the helper classes **FooSeq** (p. 1494) as well as the necessary code for DDS to manipulate the type (serialize it so that it can be sent over the network) as well as the implied **FooDataReader** (p. 1444) and **FooDataWriter** (p. 1475) types that allow the application to send and receive data of this type.

See also:

FooSeq (p. 1494), **FooDataWriter** (p. 1475), **FooDataReader** (p. 1444), **FooTypeSupport** (p. 1509), **rtiddsgen** (p. 220)

6.232 FooDataReader Struct Reference

<<***interface***>> *(*p. *199)* <<***generic***>> *(*p. *199)* User data type-specific data reader.

Inheritance diagram for FooDataReader::

Public Member Functions

- virtual **DDS_ReturnCode_t read** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

  Access a collection of data samples from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t take** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

  Access a collection of data-samples from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t read_w_condition** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)=0

  Accesses via ***FooDataReader::read*** *(*p. *1447) the samples that match the criteria specified in the* ***DDSReadCondition*** *(*p. *1374).*

- virtual **DDS_ReturnCode_t take_w_condition** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, **DDSReadCondition** *condition)=0

  Analogous to ***FooDataReader::read_w_condition*** *(*p. *1454) except it accesses samples via the* ***FooDataReader::take*** *(*p. *1448) operation.*

- virtual **DDS_ReturnCode_t read_next_sample** (**Foo** &received_data, **DDS_SampleInfo** &sample_info)=0

  Copies the next not-previously-accessed data value from the ***DDSDataReader*** *(*p. *1087).*

- virtual **DDS_ReturnCode_t take_next_sample** (**Foo** &received_data, **DDS_SampleInfo** &sample_info)=0

  Copies the next not-previously-accessed data value from the ***DDSDataReader*** *(*p. *1087).*

^ virtual **DDS_ReturnCode_t read_instance** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, const **DDS_InstanceHandle_t** &a_handle, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

*Access a collection of data samples from the **DDSDataReader*** (p. 1087).

^ virtual **DDS_ReturnCode_t take_instance** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, const **DDS_InstanceHandle_t** &a_handle, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

*Access a collection of data samples from the **DDSDataReader*** (p. 1087).

^ virtual **DDS_ReturnCode_t read_instance_w_condition** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, const **DDS_InstanceHandle_t** &previous_handle, **DDSReadCondition** *condition)=0

*Accesses via **FooDataReader::read_instance*** (p. 1459) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

^ virtual **DDS_ReturnCode_t take_instance_w_condition** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, const **DDS_InstanceHandle_t** &previous_handle, **DDSReadCondition** *condition)=0

*Accesses via **FooDataReader::take_instance*** (p. 1460) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

^ virtual **DDS_ReturnCode_t read_next_instance** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, const **DDS_InstanceHandle_t** &previous_handle, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

*Access a collection of data samples from the **DDSDataReader*** (p. 1087).

^ virtual **DDS_ReturnCode_t take_next_instance** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_Long** max_samples, const **DDS_InstanceHandle_t** &previous_handle, **DDS_SampleStateMask** sample_states, **DDS_ViewStateMask** view_states, **DDS_InstanceStateMask** instance_states)=0

*Access a collection of data samples from the **DDSDataReader*** (p. 1087).

- virtual **DDS_ReturnCode_t read_next_instance_w_condition** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_-Long** max_samples, const **DDS_InstanceHandle_t** &previous_handle, **DDSReadCondition** *condition)=0

  *Accesses via **FooDataReader::read_next_instance*** (p. 1464) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t take_next_instance_w_condition** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq, **DDS_-Long** max_samples, const **DDS_InstanceHandle_t** &previous_handle, **DDSReadCondition** *condition)=0

  *Accesses via **FooDataReader::take_next_instance*** (p. 1467) *the samples that match the criteria specified in the **DDSReadCondition*** (p. 1374).

- virtual **DDS_ReturnCode_t return_loan** (**FooSeq** &received_data, **DDS_SampleInfoSeq** &info_seq)=0

  *Indicates to the **DDSDataReader*** (p. 1087) *that the application is done accessing the collection of* `received_data` *and* `info_seq` *obtained by some earlier invocation of read or take on the **DDSDataReader*** (p. 1087).

- virtual **DDS_ReturnCode_t get_key_value** (**Foo** &key_holder, const **DDS_InstanceHandle_t** handle)=0

  Retrieve the instance `key` *that corresponds to an instance* `handle`.

- virtual **DDS_InstanceHandle_t lookup_instance** (const **Foo** &key_-holder)=0

  Retrieves the instance `handle` *that corresponds to an instance* `key_holder`.

Static Public Member Functions

- static **FooDataReader** * **narrow** (**DDSDataReader** *reader)

  *Narrow the given **DDSDataReader*** (p. 1087) *pointer to a **FooDataReader*** (p. 1444) *pointer.*

6.232.1 Detailed Description

<<***interface***>> (p. 199) <<***generic***>> (p. 199) User data type-specific data reader.

Defines the user data type specific reader interface generated for each application class.

The concrete user data type reader automatically generated by the implementation is an incarnation of this class.

See also:

DDSDataReader (p. 1087)
Foo (p. 1443)
FooDataWriter (p. 1475)
rtiddsgen (p. 220)

6.232.2 Member Function Documentation

6.232.2.1 static FooDataReader* FooDataReader::narrow (DDSDataReader * *reader*) [static]

Narrow the given **DDSDataReader** (p. 1087) pointer to a **FooDataReader** (p. 1444) pointer.

6.232.2.2 virtual DDS_ReturnCode_t FooDataReader::read (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

This operation offers the same functionality and API as **FooDataReader::take** (p. 1448) except that the samples returned remain in the **DDSDataReader** (p. 1087) such that they can be retrieved again by means of a read or take operation.

Please refer to the documentation of **FooDataReader::take()** (p. 1448) for details on the number of samples returned within the received_data and info_seq as well as the order in which the samples appear in these sequences.

The act of reading a sample changes its `sample_state` to **DDS_READ_SAMPLE_STATE** (p. 112). If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to be **DDS_NOT_NEW_VIEW_STATE** (p. 114). It will not affect the `instance_state` of the instance.

Important: If the samples "returned" by this method are loaned from RTI Connext (see **FooDataReader::take** (p. 1448) for more information on memory loaning), it is important that their contents not be changed. Because the memory in which the data is stored belongs to the middleware, any modifications made to the data will be seen the next time the same samples are read or taken; the samples will no longer reflect the state that was received from the network.

Parameters:

received_data <<*inout*>> *(p. 200)* User data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq <<*inout*>> *(p. 200)* A **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples <<*in*>> *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

sample_states <<*in*>> *(p. 200)* Data samples matching one of these `sample_states` are returned.

view_states <<*in*>> *(p. 200)* Data samples matching one of these `view_state` are returned.

instance_states <<*in*>> *(p. 200)* Data samples matching ones of these `instance_state` are returned.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read_w_condition (p. 1454), **FooDataReader::take** (p. 1448), **FooDataReader::take_w_condition** (p. 1456)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.2.3 virtual DDS_ReturnCode_t FooDataReader::take (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Access a collection of data-samples from the **DDSDataReader** (p. 1087).

The operation will return the list of samples received by the **DDSDataReader** (p. 1087) since the last **FooDataReader::take** (p. 1448) operation that match the specified **DDS_SampleStateMask** (p. 111), **DDS_ViewStateMask** (p. 113) and **DDS_InstanceStateMask** (p. 116).

This operation may fail with **DDS_RETCODE_ERROR** (p. 315) if **DDS_DataReaderResourceLimitsQosPolicy::max_outstanding_reads** (p. 526) limit has been exceeded.

The actual number of samples returned depends on the information that has been received by the middleware as well as the **DDS_HistoryQosPolicy** (p. 758), **DDS_ResourceLimitsQosPolicy** (p. 879), **DDS_DataReaderResourceLimitsQosPolicy** (p. 521) and the characteristics of the data-type that is associated with the **DDSDataReader** (p. 1087):

- In the case where the **DDS_HistoryQosPolicy::kind** (p. 760) is **DDS_KEEP_LAST_HISTORY_QOS** (p. 368), the call will return at most **DDS_HistoryQosPolicy::depth** (p. 760) samples per instance.
- The maximum number of samples returned is limited by **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881), and by **DDS_DataReaderResourceLimitsQosPolicy::max_samples_per_read** (p. 527).
- For multiple instances, the number of samples returned is additionally limited by the product (**DDS_ResourceLimitsQosPolicy::max_samples_per_instance** (p. 882) * **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882))
- If **DDS_DataReaderResourceLimitsQosPolicy::max_infos** (p. 525) is limited, the number of samples returned may also be limited if insufficient **DDS_SampleInfo** (p. 912) resources are available.

If the read or take succeeds and the number of samples returned has been limited (by means of a maximum limit, as listed above, or insufficient **DDS_SampleInfo** (p. 912) resources), the call will complete successfully and provide those samples the reader is able to return. The user may need to make additional calls, or return outstanding loaned buffers in the case of insuffifficient resources, in order to access remaining samples.

Note that in the case where the **DDSTopic** (p. 1419) associated with the **DDSDataReader** (p. 1087) is bound to a data-type that has no key definition, then there will be at most one instance in the **DDSDataReader** (p. 1087). So the per-sample limits will apply.

The act of *taking* a sample removes it from RTI Connext so it cannot be read or taken again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the sample's instance to **DDS_NOT_NEW_VIEW_STATE** (p. 114). It will not affect the `instance_state` of the sample's instance.

After **FooDataReader::take** (p. 1448) completes, `received_data` and `info_seq` will be of the same length and contain the received data.

If the sequences are empty (maximum size equals 0) when the **FooDataReader::take** (p. 1448) is called, the samples returned in the `received_data` and the corresponding `info_seq` are 'loaned' to the application from buffers

provided by the **DDSDataReader** (p. 1087). The application can use them as desired and has guaranteed exclusive access to them.

Once the application completes its use of the samples it must 'return the loan' to the **DDSDataReader** (p. 1087) by calling the **FooDataReader::return_loan** (p. 1471) operation.

Important: When you loan data from the middleware, you *must not* keep any pointers to any part of the data samples or the **DDS_SampleInfo** (p. 912) objects after the call to **FooDataReader::return_loan** (p. 1471). Returning the loan places the objects back into a pool, allowing the middleware to overwrite them with new data.

Note: While you must call **FooDataReader::return_loan** (p. 1471) at some point, you do *not* have to do so before the next **FooDataReader::take** (p. 1448) call. However, failure to return the loan will eventually deplete the **DDSDataReader** (p. 1087) of the buffers it needs to receive new samples and eventually samples will start to be lost. The total number of buffers available to the **DDSDataReader** (p. 1087) is specified by the **DDS_ResourceLimitsQosPolicy** (p. 879) and the **DDS_DataReaderResourceLimitsQosPolicy** (p. 521).

If the sequences are not empty (maximum size not equal to 0 and length not equal to 0) when **FooDataReader::take** (p. 1448) is called, samples are copied to received_data and info_seq. The application will not need to call **FooDataReader::return_loan** (p. 1471).

The order of the samples returned to the caller depends on the **DDS_PresentationQosPolicy** (p. 823).

- If **DDS_PresentationQosPolicy::access_scope** (p. 826) is **DDS_INSTANCE_PRESENTATION_QOS** (p. 352), the returned collection is a list where samples belonging to the same data instance are consecutive.

- If **DDS_PresentationQosPolicy::access_scope** (p. 826) is **DDS_TOPIC_PRESENTATION_QOS** (p. 352) and **DDS_PresentationQosPolicy::ordered_access** (p. 827) is set to **DDS_BOOLEAN_FALSE** (p. 299), then returned collection is a list where samples belonging to the same data instance are consecutive.

- If **DDS_PresentationQosPolicy::access_scope** (p. 826) is **DDS_TOPIC_PRESENTATION_QOS** (p. 352) and **DDS_PresentationQosPolicy::ordered_access** (p. 827) is set to **DDS_BOOLEAN_TRUE** (p. 298), then the returned collection is a list were the relative order of samples is preserved also accross different instances. Note that samples belonging to the same instance may or may not be

consecutive. This is because to preserve order it may be necessary to mix samples from different instances.

- If **DDS_PresentationQosPolicy::access_scope** (p. 826) is **DDS_GROUP_PRESENTATION_QOS** (p. 352) and **DDS_PresentationQosPolicy::ordered_access** (p. 827) is set to **DDS_BOOLEAN_FALSE** (p. 299), then returned collection is a list where samples belonging to the same data instance are consecutive. [**Not supported (optional)**]

- If **DDS_PresentationQosPolicy::access_scope** (p. 826) is **DDS_GROUP_PRESENTATION_QOS** (p. 352) and **DDS_PresentationQosPolicy::ordered_access** (p. 827) is set to **DDS_BOOLEAN_TRUE** (p. 298), then the returned collection contains at most one sample. The difference in this case is due to the fact that is required that the application is able to read samples belonging to different **DDSDataReader** (p. 1087) objects in a specific order. [**Not supported (optional)**]

In any case, the relative order between the samples of one instance is consistent with the **DESTINATION_ORDER** (p. 365) policy:

- If **DDS_DestinationOrderQosPolicy::kind** (p. 572) is **DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS** (p. 366), samples belonging to the same instances will appear in the relative order in which there were received (FIFO, earlier samples ahead of the later samples).

- If **DDS_DestinationOrderQosPolicy::kind** (p. 572) is **DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS** (p. 366), samples belonging to the same instances will appear in the relative order implied by the `source_timestamp` (FIFO, smaller values of `source_timestamp` ahead of the larger values).

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

In addition to the collection of samples, the read and take operations also use a collection of **DDS_SampleInfo** (p. 912) structures.

6.232.3 SEQUENCES USAGE IN TAKE AND READ

The initial (input) properties of the `received_data` and `info_seq` collections will determine the precise behavior of the read or take operation. For the purposes of this description, the collections are modeled as having these properties:

- whether the collection container owns the memory of the elements within (`owns`, see **FooSeq::has_ownership** (p. 1508))
- the current-length (`len`, see **FooSeq::length()** (p. 1501))
- the maximum length (`max_len`, see **FooSeq::maximum()** (p. 1503))

The initial values of the `owns`, `len` and `max_len` properties for the `received_data` and `info_seq` collections govern the behavior of the read and take operations as specified by the following rules:

1. The values of `owns`, `len` and `max_len` properties for the two collections must be identical. Otherwise read/take will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).
2. On successful output, the values of `owns`, `len` and `max_len` will be the same for both collections.
3. If the initial `max_len==0`, then the `received_data` and `info_seq` collections will be filled with elements that are loaned by the **DDSDataReader** (p. 1087). On output, `owns` will be FALSE, `len` will be set to the number of values returned, and `max_len` will be set to a value verifying `max_len >= len`. The use of this variant allows for zero-copy access to the data and the application will need to return the loan to the **DDSDataWriter** (p. 1113) using **FooDataReader::return_loan** (p. 1471).
4. If the initial `max_len>0` and `owns==FALSE`, then the read or take operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315). This avoids the potential hard-to-detect memory leaks caused by an application forgetting to return the loan.
5. If initial `max_len>0` and `owns==TRUE`, then the read or take operation will copy the `received_data` values and **DDS_SampleInfo** (p. 912) values into the elements already inside the collections. On output, `owns` will be TRUE, `len` will be set to the number of values copied and `max_len` will remain unchanged. The use of this variant forces a copy but the application can control where the copy is placed and the application will not need to return the loan. The number of samples copied depends on the relative values of `max_len` and `max_samples`:
   - If `max_samples` == LENGTH_UNLIMITED, then at most `max_len` values will be copied. The use of this variant lets the application limit the number of samples returned to what the sequence can accommodate.
   - If `max_samples <= max_len`, then at most `max_samples` values will be copied. The use of this variant lets the application limit the number of samples returned to fewer that what the sequence can accommodate.

- If `max_samples` > `max_len`, then the read or take operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315). This avoids the potential confusion where the application expects to be able to access up to `max_samples`, but that number can never be returned, even if they are available in the **DDSDataReader** (p. 1087), because the output sequence cannot accommodate them.

As described above, upon completion, the `received_data` and `info_seq` collections may contain elements loaned from the **DDSDataReader** (p. 1087). If this is the case, the application will need to use **FooDataReader::return_loan** (p. 1471) to return the loan once it is no longer using the `received_data` in the collection. When **FooDataReader::return_loan** (p. 1471) completes, the collection will have `owns=FALSE` and `max_len=0`. The application can determine whether it is necessary to return the loan or not based on how the state of the collections when the read/take operation was called or by accessing the `owns` property. However, in many cases it may be simpler to always call **FooDataReader::return_loan** (p. 1471), as this operation is harmless (i.e., it leaves all elements unchanged) if the collection does not have a loan.

To avoid potential memory leaks, the implementation of the **Foo** (p. 1443) and **DDS_SampleInfo** (p. 912) collections should disallow changing the length of a collection for which `owns==FALSE`. Furthermore, deleting a collection for which `owns==FALSE` should be considered an error.

On output, the collection of **Foo** (p. 1443) values and the collection of **DDS_SampleInfo** (p. 912) structures are of the same length and are in a one-to-one correspondence. Each **DDS_SampleInfo** (p. 912) provides information, such as the source_timestamp, the sample_state, view_state, and instance_state, etc., about the corresponding sample.

Some elements in the returned collection may not have valid data. If the instance_state in the **DDS_SampleInfo** (p. 912) is **DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE** (p. 117) or **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117), then the last sample for that instance in the collection (that is, the one whose **DDS_SampleInfo** (p. 912) has sample_rank==0) does not contain valid data.

Samples that contain no data do not count towards the limits imposed by the **DDS_ResourceLimitsQosPolicy** (p. 879). The act of reading/taking a sample sets its sample_state to **DDS_READ_SAMPLE_STATE** (p. 112).

If the sample belongs to the most recent generation of the instance, it will also set the view_state of the instance to **DDS_NOT_NEW_VIEW_STATE** (p. 114). It will not affect the instance_state of the instance.

This operation must be provided on the specialized class that is generated for the particular application data-type that is being read (**Foo** (p. 1443)). If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the op-

erations fails with **DDS_RETCODE_NO_DATA** (p. 316).

For an example on how `take` can be used, please refer to the **receive example** (p. 173).

Parameters:

received_data *<<inout>>* *(p. 200)* User data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

Parameters:

info_seq *<<inout>>* *(p. 200)* A **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described above.

sample_states *<<in>>* *(p. 200)* Data samples matching one of these `sample_states` are returned.

view_states *<<in>>* *(p. 200)* Data samples matching one of these `view_state` are returned.

instance_states *<<in>>* *(p. 200)* Data samples matching one of these `instance_state` are returned.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read (p. 1447)
FooDataReader::read_w_condition (p. 1454), **FooDataReader::take_w_condition** (p. 1456)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.1 virtual DDS_ReturnCode_t FooDataReader::read_w_condition (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [pure virtual]

Accesses via **FooDataReader::read** (p. 1447) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

This operation is especially useful in combination with **DDSQueryCondition** (p. 1372) to filter data samples based on the content.

The specified **DDSReadCondition** (p. 1374) must be attached to the **DDSDataReader** (p. 1087); otherwise the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

In case the **DDSReadCondition** (p. 1374) is a plain **DDSReadCondition** (p. 1374) and not the specialized **DDSQueryCondition** (p. 1372), the operation is equivalent to calling **FooDataReader::read** (p. 1447) and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in the `read_condition`. Using this operation, the application can avoid repeating the same parameters specified when creating the **DDSReadCondition** (p. 1374).

The samples are accessed with the same semantics as **FooDataReader::read** (p. 1447).

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the operation will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>> (p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>> (p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>> (p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

condition *<<in>> (p. 200)* the **DDSReadCondition** (p. 1374) to select samples of interest. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read (p. 1447)
FooDataReader::take (p. 1448), **FooDataReader::take_w_condition** (p. 1456)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.2 virtual DDS_ReturnCode_t FooDataReader::take_w_condition (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, DDSReadCondition * *condition*) [pure virtual]

Analogous to **FooDataReader::read_w_condition** (p. 1454) except it accesses samples via the **FooDataReader::take** (p. 1448) operation.

This operation is analogous to **FooDataReader::read_w_condition** (p. 1454) except that it accesses samples via the **FooDataReader::take** (p. 1448) operation.

The specified **DDSReadCondition** (p. 1374) must be attached to the **DDSDataReader** (p. 1087); otherwise the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

The samples are accessed with the same semantics as **FooDataReader::take** (p. 1448).

This operation is especially useful in combination with **DDSQueryCondition** (p. 1372) to filter data samples based on the content.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>>* *(p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>>* *(p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

condition *<<in>>* *(p. 200)* the **DDSReadCondition** (p. 1374) to select samples of interest. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read_w_condition (p. 1454), **FooDataReader::read** (p. 1447)

FooDataReader::take (p. 1448)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.3 virtual DDS_ReturnCode_t FooDataReader::read_next_sample (Foo & *received_data*, DDS_SampleInfo & *sample_info*) [pure virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

This operation copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087). This operation also copies the corresponding **DDS_SampleInfo** (p. 912). The implied order among the samples stored in the **DDSDataReader** (p. 1087) is the same as for the **FooDataReader::read** (p. 1447) operation.

The **FooDataReader::read_next_sample** (p. 1457) operation is semantically equivalent to the **FooDataReader::read** (p. 1447) operation, where the input data sequences has max_len=1, the sample_states=NOT_READ, the view_states=ANY_VIEW_STATE, and the instance_states=ANY_INSTANCE_STATE.

The **FooDataReader::read_next_sample** (p. 1457) operation provides a simplified API to 'read' samples, avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the **DDSDataReader** (p. 1087), the operation will fail with **DDS_RETCODE_NO_DATA** (p. 316) and nothing is copied.

Parameters:

received_data *<<inout>> (p. 200)* user data type-specific **Foo** (p. 1443) object where the next received data sample will be returned. The received_data must have been fully allocated. Otherwise, this operation may fail.

sample_info *<<inout>> (p. 200)* a **DDS_SampleInfo** (p. 912) object where the next received sample info will be returned.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read (p. 1447)

6.232.3.4 virtual DDS_ReturnCode_t FooDataReader::take_next_sample (Foo & *received_data*, DDS_SampleInfo & *sample_info*) [pure virtual]

Copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087).

This operation copies the next not-previously-accessed data value from the **DDSDataReader** (p. 1087) and 'removes' it from the **DDSDataReader** (p. 1087) so that it is no longer accessible. This operation also copies the corresponding **DDS_SampleInfo** (p. 912). This operation is analogous to the **FooDataReader::read_next_sample** (p. 1457) except for the fact that the sample is removed from the **DDSDataReader** (p. 1087).

The **FooDataReader::take_next_sample** (p. 1458) operation is semantically equivalent to the **FooDataReader::take** (p. 1448) operation, where the input data sequences has max_len=1, the sample_states=NOT_READ, the view_states=ANY_VIEW_STATE, and the instance_states=ANY_INSTANCE_STATE.

The **FooDataReader::read_next_sample** (p. 1457) operation provides a simplified API to 'take' samples, avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the **DDSDataReader** (p. 1087), the operation will fail with **DDS_RETCODE_NO_DATA** (p. 316) and nothing is copied.

Parameters:

received_data *<<inout>> (p. 200)* user data type-specific **Foo** (p. 1443) object where the next received data sample will be returned. The received_data must have been fully allocated. Otherwise, this operation may fail.

sample_info *<<inout>> (p. 200)* a **DDS_SampleInfo** (p. 912) object where the next received sample info will be returned.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::take (p. 1448)

6.232.3.5 virtual DDS_ReturnCode_t FooDataReader::read_instance (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *a_handle*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087). The behavior is identical to **FooDataReader::read** (p. 1447), except that all samples returned belong to the single specified instance whose handle is `a_handle`.

Upon successful completion, the data collection will contain samples all belonging to the same instance. The corresponding **DDS_SampleInfo** (p. 912) verifies **DDS_SampleInfo::instance_handle** (p. 917) == `a_handle`.

The **FooDataReader::read_instance** (p. 1459) operation is semantically equivalent to the **FooDataReader::read** (p. 1447) operation, except in building the collection, the **DDSDataReader** (p. 1087) will check that the sample belongs to the specified instance and otherwise it will not place the sample in the returned collection.

The behavior of the **FooDataReader::read_instance** (p. 1459) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::read_instance** (p. 1459) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

This operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) if the **DDS_InstanceHandle_t** (p. 53) `a_handle` does not correspond to an existing data-object known to the **DDSDataReader** (p. 1087).

Parameters:

received_data *<<inout>> (p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>> (p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

a_handle *<<in>>* *(p. 200)* The specified instance to return samples for. The method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) if the `handle` does not correspond to an existing data-object known to the **DDSDataReader** (p. 1087).

sample_states *<<in>>* *(p. 200)* data samples matching ones of these `sample_states` are returned

view_states *<<in>>* *(p. 200)* data samples matching ones of these `view_state` are returned

instance_states *<<in>>* *(p. 200)* data samples matching ones of these `instance_state` are returned

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read (p. 1447)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.6 virtual DDS_ReturnCode_t FooDataReader::take_instance (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *a_handle*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087). The behavior is identical to **FooDataReader::take** (p. 1448), except for that all samples returned belong to the single specified instance whose handle is `a_handle`.

The semantics are the same for the **FooDataReader::take** (p. 1448) operation, except in building the collection, the **DDSDataReader** (p. 1087) will check that the sample belongs to the specified instance, and otherwise it will not place the sample in the returned collection.

The behavior of the **FooDataReader::take_instance** (p. 1460) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::take_instance** (p. 1460) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method fails with **DDS_RETCODE_NO_DATA** (p. 316).

This operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) if the **DDS_InstanceHandle_t** (p. 53) `a_handle` does not correspond to an existing data-object known to the **DDSDataReader** (p. 1087).

Parameters:

received_data *<<inout>>* *(p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>>* *(p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

a_handle *<<in>>* *(p. 200)* The specified instance to return samples for. The method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) if the `handle` does not correspond to an existing data-object known to the **DDSDataReader** (p. 1087).

sample_states *<<in>>* *(p. 200)* data samples matching ones of these `sample_states` are returned

view_states *<<in>>* *(p. 200)* data samples matching ones of these `view_state` are returned

instance_states *<<in>>* *(p. 200)* data samples matching ones of these `instance_state` are returned

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::take (p. 1448)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.7 virtual DDS_ReturnCode_t FooDataReader::read_instance_w_condition (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *previous_handle*, DDSReadCondition * *condition*) [pure virtual]

Accesses via **FooDataReader::read_instance** (p. 1459) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087). The behavior is identical to **FooDataReader::read_instance** (p. 1459), except that all returned samples satisfy the specified condition. In other words, on success, all returned samples belong to belong the single specified instance whose handle is `a_handle`, and for which the specified **DDSReadCondition** (p. 1374) evaluates to TRUE.

The behavior of the **FooDataReader::read_instance_w_condition** (p. 1462) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::read_instance_w_condition** (p. 1462) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>> (p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>> (p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>> (p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

previous_handle *<<in>>* *(p. 200)* The 'next smallest' instance with a value greater than this value that has available samples will be returned.

condition *<<in>>* *(p. 200)* the **DDSReadCondition** (p. 1374) to select samples of interest. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read_next_instance (p. 1464)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.8 virtual DDS_ReturnCode_t FooDataReader::take_instance_w_condition (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *previous_handle*, DDSReadCondition * *condition*) [pure virtual]

Accesses via **FooDataReader::take_instance** (p. 1460) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087) and 'removes' them from the **DDSDataReader** (p. 1087). The behavior is identical to **FooDataReader::take_instance** (p. 1460), except that all returned samples satisfy the specified condition. In other words, on success, all returned samples belong to belong the single specified instance whose handle is `a_handle`, and for which the specified **DDSReadCondition** (p. 1374) evaluates to TRUE.

The operation has the same behavior as **FooDataReader::read_instance_w_condition** (p. 1462), except that the samples are 'taken' from the **DDSDataReader** (p. 1087) such that they are no longer accessible via subsequent 'read' or 'take' operations.

The behavior of the **FooDataReader::take_instance_w_condition** (p. 1463) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::take_instance_w_condition** (p. 1463) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>>* *(p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>>* *(p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

previous_handle *<<in>>* *(p. 200)* The 'next smallest' instance with a value greater than this value that has available samples will be returned.

condition *<<in>>* *(p. 200)* the **DDSReadCondition** (p. 1374) to select samples of interest. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), or **DDS_RETCODE_NO_DATA** (p. 316), **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::take_next_instance (p. 1467)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.9 virtual DDS_ReturnCode_t FooDataReader::read_next_instance (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *previous_handle*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087) where all the samples belong to a single instance. The behavior is similar to **FooDataReader::read_instance** (p. 1459), except that the actual instance is not directly specified. Rather, the samples will all belong to the 'next' instance with `instance_handle` 'greater' than the specified 'previous_handle' that has available samples.

This operation implies the existence of a total order 'greater-than' relationship between the instance handles. The specifics of this relationship are not all important and are implementation specific. The important thing is that, according to the middleware, all instances are ordered relative to each other. This ordering is between the instance handles; It should not depend on the state of the instance (e.g. whether it has data or not) and must be defined even for instance handles that do not correspond to instances currently managed by the **DDSDataReader** (p. 1087). For the purposes of the ordering, it should be 'as if' each instance handle was represented as unique integer.

The behavior of **FooDataReader::read_next_instance** (p. 1464) is 'as if' the **DDSDataReader** (p. 1087) invoked **FooDataReader::read_instance** (p. 1459), passing the smallest `instance_handle` among all the ones that: (a) are greater than `previous_handle`, and (b) have available samples (i.e. samples that meet the constraints imposed by the specified states).

The special value **DDS_HANDLE_NIL** (p. 55) is guaranteed to be 'less than' any valid `instance_handle`. So the use of the parameter value `previous_handle` == **DDS_HANDLE_NIL** (p. 55) will return the samples for the instance which has the smallest `instance_handle` among all the instances that contain available samples.

The operation **FooDataReader::read_next_instance** (p. 1464) is intended to be used in an application-driven iteration, where the application starts by passing `previous_handle` == **DDS_HANDLE_NIL** (p. 55), examines the samples returned, and then uses the `instance_handle` returned in the **DDS_SampleInfo** (p. 912) as the value of the `previous_handle` argument to the next call to **FooDataReader::read_next_instance** (p. 1464). The iteration continues until **FooDataReader::read_next_instance** (p. 1464) fails with the value **DDS_RETCODE_NO_DATA** (p. 316).

Note that it is possible to call the **FooDataReader::read_next_instance** (p. 1464) operation with a `previous_handle` that does not correspond to an instance currently managed by the **DDSDataReader** (p. 1087). This is because as stated earlier the 'greater-than' relationship is defined even for handles not managed by the **DDSDataReader** (p. 1087). One practical situation where this may occur is when an application is iterating though all the instances, takes all the samples of a **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117) instance, returns the loan (at which point the instance information may be removed, and thus the handle becomes invalid), and tries to read the next instance.

The behavior of the **FooDataReader::read_next_instance** (p. 1464) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::read_instance** (p. 1459) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>>* *(p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>>* *(p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

previous_handle *<<in>>* *(p. 200)* The 'next smallest' instance with a value greater than this value that has available samples will be returned.

sample_states *<<in>>* *(p. 200)* data samples matching ones of these `sample_states` are returned

view_states *<<in>>* *(p. 200)* data samples matching ones of these `view_state` are returned

instance_states *<<in>>* *(p. 200)* data samples matching ones of these `instance_state` are returned

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read (p. 1447)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.10 virtual DDS_ReturnCode_t FooDataReader::take_next_instance (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *previous_handle*, DDS_SampleStateMask *sample_states*, DDS_ViewStateMask *view_states*, DDS_InstanceStateMask *instance_states*) [pure virtual]

Access a collection of data samples from the **DDSDataReader** (p. 1087).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087) and 'removes' them from the **DDSDataReader** (p. 1087).

This operation has the same behavior as **FooDataReader::read_next_instance** (p. 1464), except that the samples are 'taken' from the **DDSDataReader** (p. 1087) such that they are no longer accessible via subsequent 'read' or 'take' operations.

Similar to the operation **FooDataReader::read_next_instance** (p. 1464), it is possible to call **FooDataReader::take_next_instance** (p. 1467) with a `previous_handle` that does not correspond to an instance currently managed by the **DDSDataReader** (p. 1087).

The behavior of the **FooDataReader::take_next_instance** (p. 1467) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::take_next_instance** (p. 1467) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>>* *(p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>>* *(p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **Foo-**

DataReader::take() (p. 1448).

previous_handle *<<in>>* *(p. 200)* The 'next smallest' instance with a value greater than this value that has available samples will be returned.

sample_states *<<in>>* *(p. 200)* data samples matching ones of these `sample_states` are returned

view_states *<<in>>* *(p. 200)* data samples matching ones of these `view_state` are returned

instance_states *<<in>>* *(p. 200)* data samples matching ones of these `instance_state` are returned

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::take (p. 1448)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.11 virtual DDS_ReturnCode_t FooDataReader::read_next_instance_w_condition (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *previous_handle*, DDSReadCondition * *condition*) [pure virtual]

Accesses via **FooDataReader::read_next_instance** (p. 1464) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087). The behavior is identical to **FooDataReader::read_next_instance** (p. 1464), except that all returned samples satisfy the specified condition. In other words, on success, all returned samples belong to the same instance, and the instance is the instance with 'smallest' `instance_handle` among the ones that verify: (a) `instance_handle` >= `previous_handle`, and (b) have samples for which the specified **DDSReadCondition** (p. 1374) evaluates to TRUE.

Similar to the operation **FooDataReader::read_next_instance** (p. 1464), it is possible to call **FooDataReader::read_next_instance_w_condition** (p. 1468) with a `previous_handle` that does not correspond to an instance currently managed by the **DDSDataReader** (p. 1087).

The behavior of the **FooDataReader::read_next_instance_w_condition** (p. 1468) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to the **FooDataReader::read** (p. 1447), the **FooDataReader::read_next_instance_w_condition** (p. 1468) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>> (p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>> (p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>> (p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

previous_handle *<<in>> (p. 200)* The 'next smallest' instance with a value greater than this value that has available samples will be returned.

condition *<<in>> (p. 200)* the **DDSReadCondition** (p. 1374) to select samples of interest. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) **DDS_RETCODE_NO_DATA** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::read_next_instance (p. 1464)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.12 virtual DDS_ReturnCode_t FooDataReader::take_next_instance_w_condition (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*, DDS_Long *max_samples*, const DDS_InstanceHandle_t & *previous_handle*, DDSReadCondition * *condition*) [pure virtual]

Accesses via **FooDataReader::take_next_instance** (p. 1467) the samples that match the criteria specified in the **DDSReadCondition** (p. 1374).

This operation accesses a collection of data values from the **DDSDataReader** (p. 1087) and 'removes' them from the **DDSDataReader** (p. 1087).

The operation has the same behavior as **FooDataReader::read_next_instance_w_condition** (p. 1468), except that the samples are 'taken' from the **DDSDataReader** (p. 1087) such that they are no longer accessible via subsequent 'read' or 'take' operations.

Similar to the operation **FooDataReader::read_next_instance** (p. 1464), it is possible to call **FooDataReader::take_next_instance_w_condition** (p. 1470) with a `previous_handle` that does not correspond to an instance currently managed by the **DDSDataReader** (p. 1087).

The behavior of the **FooDataReader::take_next_instance_w_condition** (p. 1470) operation follows the same rules as the **FooDataReader::read** (p. 1447) operation regarding the pre-conditions and post-conditions for the `received_data` and `sample_info`. Similar to **FooDataReader::read** (p. 1447), the **FooDataReader::take_next_instance_w_condition** (p. 1470) operation may 'loan' elements to the output collections, which must then be returned by means of **FooDataReader::return_loan** (p. 1471).

Similar to the **FooDataReader::read** (p. 1447), this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the **DDSDataReader** (p. 1087) has no samples that meet the constraints, the method will fail with **DDS_RETCODE_NO_DATA** (p. 316).

Parameters:

received_data *<<inout>>* *(p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples will be returned.

info_seq *<<inout>>* *(p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info will be returned.

max_samples *<<in>>* *(p. 200)* The maximum number of samples to be returned. If the special value **DDS_LENGTH_UNLIMITED** (p. 371) is provided, as many samples will be returned as are available, up to the limits described in the documentation for **FooDataReader::take()** (p. 1448).

previous_handle *<<in>>* *(p. 200)* The 'next smallest' instance with a value greater than this value that has available samples will be returned.

condition *<<in>>* *(p. 200)* the **DDSReadCondition** (p. 1374) to select samples of interest. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), or **DDS_RETCODE_NO_DATA** (p. 316), **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::take_next_instance (p. 1467)
DDS_LENGTH_UNLIMITED (p. 371)

6.232.3.13 virtual DDS_ReturnCode_t FooDataReader::return_loan (FooSeq & *received_data*, DDS_SampleInfoSeq & *info_seq*) [pure virtual]

Indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of `received_data` and `info_seq` obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).

This operation indicates to the **DDSDataReader** (p. 1087) that the application is done accessing the collection of `received_data` and `info_seq` obtained by some earlier invocation of read or take on the **DDSDataReader** (p. 1087).

The `received_data` and `info_seq` must belong to a single related "pair"; that is, they should correspond to a pair returned from a single call to read or take. The `received_data` and `info_seq` must also have been obtained from the same **DDSDataReader** (p. 1087) to which they are returned. If either of these conditions is not met, the operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

The operation **FooDataReader::return_loan** (p. 1471) allows implementations of the read and take operations to "loan" buffers from the **DDSDataReader** (p. 1087) to the application and in this manner provide "zero-copy" access to the data. During the loan, the **DDSDataReader** (p. 1087) will guarantee that the data and sample-information are not modified.

It is not necessary for an application to return the loans immediately after the read or take calls. However, as these buffers correspond to internal resources inside the **DDSDataReader** (p. 1087), the application should not retain them indefinitely.

The use of **FooDataReader::return_loan** (p. 1471) is only necessary if the read or take calls "loaned" buffers to the application. This only occurs if the `received_data` and `info_Seq` collections had `max_len=0` at the time read or take was called.

The application may also examine the "owns" property of the collection to determine where there is an outstanding loan. However, calling **FooDataReader::return_loan** (p. 1471) on a collection that does not have a loan is safe and has no side effects.

If the collections had a loan, upon completion of **FooDataReader::return_loan** (p. 1471), the collections will have `max_len=0`.

Similar to read, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

Parameters:

received_data *<<in>> (p. 200)* user data type-specific **FooSeq** (p. 1494) object where the received data samples was obtained from earlier invocation of read or take on the **DDSDataReader** (p. 1087).

Parameters:

info_seq *<<in>> (p. 200)* a **DDS_SampleInfoSeq** (p. 922) object where the received sample info was obtained from earlier invocation of read or take on the **DDSDataReader** (p. 1087).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

6.232.3.14 virtual DDS_ReturnCode_t FooDataReader::get_key_value (Foo & *key_holder*, const DDS_InstanceHandle_t *handle*) [pure virtual]

Retrieve the instance `key` that corresponds to an instance `handle`.

Useful for keyed data types.

The operation will only fill the fields that form the `key` inside the `key_holder` instance.

For keyed data types, this operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) if the `handle` does not correspond to an existing data-object known to the **DDSDataReader** (p. 1087).

Parameters:

key_holder *<<inout>>* *(p. 200)* a user data type specific key holder, whose `key` fields are filled by this operation. If **Foo** (p. 1443) has no key, this method has no effect.

handle *<<in>>* *(p. 200)* the `instance` whose key is to be retrieved. If **Foo** (p. 1443) *has* a key, `handle` must represent an existing instance of type **Foo** (p. 1443) known to the **DDSDataReader** (p. 1087). Otherwise, this method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315). If **Foo** (p. 1443) *has* a key and `handle` is **DDS_HANDLE_NIL** (p. 55), this method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315). If **Foo** (p. 1443) has a key and `handle` represents an instance of another type or an instance of type **Foo** (p. 1443) that has been unregistered, this method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315). If **Foo** (p. 1443) has no key, this method has no effect.

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataWriter::get_key_value (p. 1492)

6.232.3.15 virtual DDS_InstanceHandle_t FooDataReader::lookup_instance (const Foo & *key_holder*) [pure virtual]

Retrieves the instance `handle` that corresponds to an instance `key_holder`.

Useful for keyed data types.

This operation takes as a parameter an instance and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The instance parameter is only used for the purpose of examining the fields that define the key. This operation does not register the instance in question. If the instance has not been previously registered, or if for any other reason the Service is unable to provide an instance handle, the Service will return the special value HANDLE_NIL.

Parameters:

key_holder *<<in>>* *(p. 200)* a user data type specific key holder.

Returns:

the instance handle associated with this instance. If **Foo** (p. 1443) has no key, this method has no effect and returns **DDS_HANDLE_NIL** (p. 55)

6.233 FooDataWriter Struct Reference

<<interface>> (p. 199) *<<generic>>* (p. 199) User data type specific data writer.

Inheritance diagram for FooDataWriter::

Public Member Functions

- virtual **DDS_InstanceHandle_t register_instance** (const **Foo** &instance_data)=0

  Informs RTI Connext that the application will be modifying a particular instance.

- virtual **DDS_InstanceHandle_t register_instance_w_timestamp** (const **Foo** &instance_data, const **DDS_Time_t** &source_timestamp)=0

  Performs the same functions as register_instance except that the application provides the value for the `source_timestamp`.

- virtual **DDS_InstanceHandle_t register_instance_w_params** (const **Foo** &instance_data, **DDS_WriteParams_t** &params)=0

  *Performs the same function as **FooDataWriter::register_instance*** (p. 1478) *and **FooDataWriter::register_instance_w_timestamp*** (p. 1479) *except that it also provides the values contained in* `params`.

- virtual **DDS_ReturnCode_t unregister_instance** (const **Foo** &instance_data, const **DDS_InstanceHandle_t** &handle)=0

  *Reverses the action of **FooDataWriter::register_instance*** (p. 1478).

- virtual **DDS_ReturnCode_t unregister_instance_w_timestamp** (const **Foo** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)=0

  *Performs the same function as **FooDataWriter::unregister_instance*** (p. 1480) *except that it also provides the value for the* `source_timestamp`.

- virtual **DDS_ReturnCode_t unregister_instance_w_params** (const **Foo** &instance_data, **DDS_WriteParams_t** &params)=0

  *Performs the same function as **FooDataWriter::unregister_instance*** (p. 1480) *and **FooDataWriter::unregister_instance_w_timestamp*** (p. 1482) *except that it also provides the values contained in* `params`.

^ virtual **DDS_ReturnCode_t write** (const **Foo** &instance_data, const **DDS_InstanceHandle_t** &handle)=0

Modifies the value of a data instance.

^ virtual **DDS_ReturnCode_t write_w_timestamp** (const **Foo** &instance_data, const **DDS_InstanceHandle_t** &handle, const **DDS_Time_t** &source_timestamp)=0

*Performs the same function as **FooDataWriter::write** (p. 1484) except that it also provides the value for the* `source_timestamp`.

^ virtual **DDS_ReturnCode_t write_w_params** (const **Foo** &instance_data, **DDS_WriteParams_t** &params)=0

*Performs the same function as **FooDataWriter::write** (p. 1484) and **FooDataWriter::write_w_timestamp** (p. 1486) except that it also provides the values contained in* `params`.

^ virtual **DDS_ReturnCode_t dispose** (const **Foo** &instance_data, const **DDS_InstanceHandle_t** &instance_handle)=0

Requests the middleware to delete the data.

^ virtual **DDS_ReturnCode_t dispose_w_timestamp** (const **Foo** &instance_data, const **DDS_InstanceHandle_t** &instance_handle, const **DDS_Time_t** &source_timestamp)=0

Performs the same functions as dispose except that the application provides the value for the `source_timestamp` *that is made available to **DDSDataReader** (p. 1087) objects by means of the* `source_timestamp` *attribute inside the **DDS_SampleInfo** (p. 912).*

^ virtual **DDS_ReturnCode_t dispose_w_params** (const **Foo** &instance_data, **DDS_WriteParams_t** &params)=0

*Performs the same function as **FooDataWriter::dispose** (p. 1489) and **FooDataWriter::dispose_w_timestamp** (p. 1490) except that it also provides the values contained in* `params`.

^ virtual **DDS_ReturnCode_t get_key_value** (**Foo** &key_holder, const **DDS_InstanceHandle_t** &handle)=0

Retrieve the instance `key` *that corresponds to an instance* `handle`.

^ virtual **DDS_InstanceHandle_t lookup_instance** (const **Foo** &key_holder)=0

Retrieve the instance `handle` *that corresponds to an instance* `key_holder`.

Static Public Member Functions

- static **FooDataWriter** * **narrow** (**DDSDataWriter** *writer)

  *Narrow the given **DDSDataWriter** (p. 1113) pointer to a **FooDataWriter** (p. 1475) pointer.*

6.233.1 Detailed Description

*<<**interface**>> (p. 199) <<**generic**>> (p. 199)* User data type specific data writer.

Defines the user data type specific writer interface generated for each application class.

The concrete user data type writer automatically generated by the implementation is an incarnation of this class.

See also:

DDSDataWriter (p. 1113)
Foo (p. 1443)
FooDataReader (p. 1444)
rtiddsgen (p. 220)

6.233.2 Member Function Documentation

6.233.2.1 static FooDataWriter* FooDataWriter::narrow (DDSDataWriter * *writer*) [static]

Narrow the given **DDSDataWriter** (p. 1113) pointer to a **FooDataWriter** (p. 1475) pointer.

Check if the given `writer` is of type **FooDataWriter** (p. 1475).

Parameters:

writer *<<**in**>> (p. 200)* Base-class **DDSDataWriter** (p. 1113) to be converted to the auto-generated class **FooDataWriter** (p. 1475) that extends **DDSDataWriter** (p. 1113).

Returns:

FooDataWriter (p. 1475) if `writer` is of type **Foo** (p. 1443). Return NULL otherwise.

6.233.2.2 virtual DDS_InstanceHandle_t FooDataWriter::register_instance (const Foo & *instance_data*) [pure virtual]

Informs RTI Connext that the application will be modifying a particular instance.

This operation is only useful for keyed data types. Using it for non-keyed types causes no effect and returns **DDS_HANDLE_NIL** (p. 55). The operation takes as a parameter an instance (of which only the key value is examined) and returns a `handle` that can be used in successive **write()** (p. 1484) or **dispose()** (p. 1489) operations.

The operation gives RTI Connext an opportunity to pre-configure itself to improve performance.

The use of this operation by an application is optional even for keyed types. If an instance has not been pre-registered, the application can use the special value **DDS_HANDLE_NIL** (p. 55) as the **DDS_InstanceHandle_t** (p. 53) paramater to the write or dispose operation and RTI Connext will auto-register the instance.

For best performance, the operation should be invoked prior to calling any operation that modifies the instance, such as **FooDataWriter::write** (p. 1484), **FooDataWriter::write_w_timestamp** (p. 1486), **FooDataWriter::dispose** (p. 1489) and **FooDataWriter::dispose_w_timestamp** (p. 1490) and the handle used in conjunction with the data for those calls.

When this operation is used, RTI Connext will automatically supply the value of the `source_timestamp` that is used.

This operation may fail and return **DDS_HANDLE_NIL** (p. 55) if **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882) limit has been exceeded.

The operation is **idempotent**. If it is called for an already registered instance, it just returns the already allocated handle. This may be used to lookup and retrieve the handle allocated to a given instance.

This operation can only be called after **DDSDataWriter** (p. 1113) has been enabled. Otherwise, **DDS_HANDLE_NIL** (p. 55) will be returned.

Parameters:

instance_data *<<in>> (p. 200)* The instance that should be registered. Of this instance, only the fields that represent the key are examined by the function. .

Returns:

For keyed data type, a handle that can be used in the calls that take

a **DDS_InstanceHandle_t** (p. 53), such as write, dispose, unregister_instance, or return **DDS_HANDLE_NIL** (p. 55) on failure. If the instance_data is of a data type that has no keys, this function always return **DDS_HANDLE_NIL** (p. 55).

See also:

FooDataWriter::unregister_instance (p. 1480), **FooDataWriter::get_key_value** (p. 1492), **RELATIONSHIP BETWEEN REGISTRATION, LIVELINESS and OWNERSHIP** (p. 809)

6.233.2.3 virtual DDS_InstanceHandle_t FooDataWriter::register_instance_w_timestamp (const Foo & *instance_data*, const DDS_Time_t & *source_timestamp*) [pure virtual]

Performs the same functions as register_instance except that the application provides the value for the `source_timestamp`.

The provided `source_timestamp` potentially affects the relative order in which readers observe events from multiple writers. Refer to **DESTINATION_ORDER** (p. 365) QoS policy for details.

This operation may fail and return **DDS_HANDLE_NIL** (p. 55) if **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882) limit has been exceeded.

This operation can only be called after **DDSDataWriter** (p. 1113) has been enabled. Otherwise, **DDS_HANDLE_NIL** (p. 55) will be returned.

Parameters:

instance_data *<<in>> (p. 200)* The instance that should be registered. Of this instance, only the fields that represent the key are examined by the function.

source_timestamp *<<in>> (p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation (used in a *register*, *unregister*, *dispose*, or *write*, with either the automatically supplied timestamp or the application provided timestamp). This timestamp may potentially affect the order in which readers observe events from multiple writers.

Returns:

For keyed data type, return a handle that can be used in the calls that take a **DDS_InstanceHandle_t** (p. 53), such as write, dispose, unregister_instance, or return **DDS_HANDLE_NIL** (p. 55) on failure. If the in-

stance_data is of a data type that has no keys, this function always return **DDS_HANDLE_NIL** (p. 55).

See also:

FooDataWriter::unregister_instance (p. 1480), **FooDataWriter::get_key_value** (p. 1492)

6.233.2.4 virtual DDS_InstanceHandle_t FooDataWriter::register_instance_w_params (const Foo & *instance_data*, DDS_WriteParams_t & *params*) [pure virtual]

Performs the same function as **FooDataWriter::register_instance** (p. 1478) and **FooDataWriter::register_instance_w_timestamp** (p. 1479) except that it also provides the values contained in `params`.

See also:

FooDataWriter::write_w_params (p. 1487)

6.233.2.5 virtual DDS_ReturnCode_t FooDataWriter::unregister_instance (const Foo & *instance_data*, const DDS_InstanceHandle_t & *handle*) [pure virtual]

Reverses the action of **FooDataWriter::register_instance** (p. 1478).

This operation is useful only for keyed data types. Using it for non-keyed types causes no effect and reports no error. The operation takes as a parameter an instance (of which only the key value is examined) and a handle.

This operation should only be called on an instance that is currently registered. This includes instances that have been auto-registered by calling operations such as write or dispose as described in **FooDataWriter::register_instance** (p. 1478). Otherwise, this operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

This only need be called just once per instance, regardless of how many times register_instance was called for that instance.

When this operation is used, RTI Connext will automatically supply the value of the `source_timestamp` that is used.

This operation informs RTI Connext that the **DDSDataWriter** (p. 1113) is no longer going to provide any information about the instance. This operation also indicates that RTI Connext can locally remove all information regarding that

instance. The application should not attempt to use the `handle` previously allocated to that instance after calling **FooDataWriter::unregister_instance()** (p. 1480).

The special value **DDS_HANDLE_NIL** (p. 55) can be used for the parameter `handle`. This indicates that the identity of the instance should be automatically deduced from the `instance_data` (by means of the `key`).

If `handle` is any value other than **DDS_HANDLE_NIL** (p. 55), then it must correspond to an instance that has been registered. If there is no correspondence, the operation will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

RTI Connext will not detect the error when the `handle` is any value other than **DDS_HANDLE_NIL** (p. 55), corresponds to an instance that has been registered, but does not correspond to the instance deduced from the `instance_data` (by means of the `key`). RTI Connext will treat as if the **unregister_instance()** (p. 1480) operation is for the instance as indicated by the `handle`.

If after a **FooDataWriter::unregister_instance** (p. 1480), the application wants to modify (**FooDataWriter::write** (p. 1484) or **FooDataWriter::dispose** (p. 1489)) an instance, it has to register it again, or else use the special `handle` value **DDS_HANDLE_NIL** (p. 55).

This operation does not indicate that the instance is deleted (that is the purpose of **FooDataWriter::dispose** (p. 1489)). The operation **FooDataWriter::unregister_instance** (p. 1480) just indicates that the **DDSDataWriter** (p. 1113) no longer has anything to say about the instance. **DDSDataReader** (p. 1087) entities that are reading the instance may receive a sample with **DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE** (p. 117) for the instance, unless there are other **DDSDataWriter** (p. 1113) objects writing that same instance.

This operation can affect the ownership of the data instance (see **OWNERSHIP** (p. 355)). If the **DDSDataWriter** (p. 1113) was the exclusive owner of the instance, then calling **unregister_instance()** (p. 1480) will relinquish that ownership.

If **DDS_ReliabilityQosPolicy::kind** (p. 868) is set to **DDS_RELIABLE_RELIABILITY_QOS** (p. 363) and the unregistration would overflow the resource limits of this writer or of a reader, this operation may block for up to **DDS_ReliabilityQosPolicy::max_blocking_time** (p. 868); if this writer is still unable to unregister after that period, this method will fail with **DDS_RETCODE_TIMEOUT** (p. 316).

Parameters:

instance_data *<<in>> (p. 200)* The instance that should be unregistered. If **Foo** (p. 1443) *has* a key and `instance_handle` is **DDS_HANDLE_NIL** (p. 55), only the fields that represent the key are

examined by the function. Otherwise, `instance_data` is not used. If `instance_data` is used, it must represent an instance that has been registerd. Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) .

handle *<<in>> (p. 200)* represents the `instance` to be unregistered. If **Foo** (p. 1443) *has* a key and `handle` is **DDS_HANDLE_NIL** (p. 55), `handle` is not used and `instance` is deduced from `instance_data`. If **Foo** (p. 1443) has no key, `handle` is not used. If `handle` is used, it must represent an instance that has been registered. Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315)

See also:

FooDataWriter::register_instance (p. 1478)
FooDataWriter::unregister_instance_w_timestamp (p. 1482)
FooDataWriter::get_key_value (p. 1492)
RELATIONSHIP BETWEEN REGISTRATION, LIVELINESS and OWNERSHIP (p. 809)

6.233.2.6 virtual DDS_ReturnCode_t FooDataWriter::unregister_instance_w_timestamp (const Foo & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [pure virtual]

Performs the same function as **FooDataWriter::unregister_instance** (p. 1480) except that it also provides the value for the `source_timestamp`.

The provided `source_timestamp` potentially affects the relative order in which readers observe events from multiple writers. Refer to **DESTINATION_ORDER** (p. 365) QoS policy for details.

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the **FooDataWriter::unregister_instance** (p. 1480) operation.

This operation may block and may time out (**DDS_RETCODE_TIMEOUT** (p. 316)) under the same circumtances described for the unregister_instance operation.

Parameters:

instance_data *<<in>> (p. 200)* The instance that should be unregistered. If **Foo** (p. 1443) *has* a key and `instance_handle` is **DDS_**

HANDLE_NIL (p. 55), only the fields that represent the key are examined by the function. Otherwise, `instance_data` is not used. If `instance_data` is used, it must represent an instance that has been registerd. Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

handle *<<in>>* *(p. 200)* represents the `instance` to be unregistered. If **Foo** (p. 1443) *has* a key and `handle` is **DDS_HANDLE_NIL** (p. 55), `handle` is not used and `instance` is deduced from `instance_data`. If **Foo** (p. 1443) has no key, `handle` is not used. If `handle` is used, it must represent an instance that has been registered. Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

source_timestamp *<<in>>* *(p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation (used in a *register*, *unregister*, *dispose*, or *write*, with either the automatically supplied timestamp or the application provided timestamp). This timestamp may potentially affect the order in which readers observe events from multiple writers.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataWriter::register_instance (p. 1478)
FooDataWriter::unregister_instance (p. 1480)
FooDataWriter::get_key_value (p. 1492)

6.233.2.7 virtual DDS_ReturnCode_t FooDataWriter::unregister_instance_w_params (const Foo & *instance_data*, DDS_WriteParams_t & *params*) [pure virtual]

Performs the same function as **FooDataWriter::unregister_instance** (p. 1480) and **FooDataWriter::unregister_instance_w_timestamp** (p. 1482) except that it also provides the values contained in `params`.

See also:

FooDataWriter::write_w_params (p. 1487)
FooDataWriter::dispose_w_params (p. 1491)

6.233.2.8 virtual DDS_ReturnCode_t FooDataWriter::write (const Foo & *instance_data*, const DDS_InstanceHandle_t & *handle*) [pure virtual]

Modifies the value of a data instance.

When this operation is used, RTI Connext will automatically supply the value of the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912). (Refer to **DDS_SampleInfo** (p. 912) and **DESTINATION_ORDER** (p. 365) QoS policy for details).

As a side effect, this operation asserts liveliness on the **DDSDataWriter** (p. 1113) itself, the **DDSPublisher** (p. 1346) and the **DDSDomainParticipant** (p. 1139).

Note that the special value **DDS_HANDLE_NIL** (p. 55) can be used for the parameter `handle`. This indicates the identity of the instance should be automatically deduced from the `instance_data` (by means of the `key`).

If `handle` is any value other than **DDS_HANDLE_NIL** (p. 55), then it must correspond to an instance that has been registered. If there is no correspondence, the operation will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

RTI Connext will not detect the error when the `handle` is any value other than **DDS_HANDLE_NIL** (p. 55), corresponds to an instance that has been registered, but does not correspond to the instance deduced from the `instance_data` (by means of the `key`). RTI Connext will treat as if the **write()** (p. 1484) operation is for the instance as indicated by the `handle`.

This operation may block if the **RELIABILITY** (p. 362) `kind` is set to **DDS_RELIABLE_RELIABILITY_QOS** (p. 363) and the modification would cause data to be lost or else cause one of the limits specified in the **RESOURCE_LIMITS** (p. 371) to be exceeded.

Specifically, this operation may block in the following situations (note that the list may not be exhaustive), even if its **DDS_HistoryQosPolicyKind** (p. 368) is **DDS_KEEP_LAST_HISTORY_QOS** (p. 368):

- If (**DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) < **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882) * **DDS_HistoryQosPolicy::depth** (p. 760)), then in the situation where the `max_samples` resource limit is exhausted, RTI Connext is allowed to discard samples of some other instance, as long as at least one sample remains for such an instance. If it is still not possible to make space available to store the modification, the writer is allowed to block.
- If (**DDS_ResourceLimitsQosPolicy::max_samples** (p. 881) < **DDS_ResourceLimitsQosPolicy::max_instances** (p. 882)), then

the **DDSDataWriter** (p. 1113) may block regardless of the **DDS_HistoryQosPolicy::depth** (p. 760).

- If (**DDS_RtpsReliableWriterProtocol_t::min_send_window_size** (p. 901) < **DDS_ResourceLimitsQosPolicy::max_samples** (p. 881)), then it is possible for the send_window_size limit to be reached before RTI Connext is allowed to discard samples, in which case the **DDSDataWriter** (p. 1113) will block.

This operation may also block when using **DDS_BEST_EFFORT_RELIABILITY_QOS** (p. 363) and **DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS** (p. 422). In this case, the **DDSDataWriter** (p. 1113) will queue samples until they are sent by the asynchronous publishing thread. The number of samples that can be stored is determined by the **DDS_HistoryQosPolicy** (p. 758). If the asynchronous thread does not send samples fast enough (e.g., when using a slow **DDSFlowController** (p. 1259)), the queue may fill up. In that case, subsequent write calls will block.

If this operation *does* block for any of the above reasons, the **RELIABILITY** (p. 362) `max_blocking_time` configures the maximum time the write operation may block (waiting for space to become available). If `max_blocking_time` elapses before the **DDSDataWriter** (p. 1113) is able to store the modification without exceeding the limits, the operation will time out (**DDS_RETCODE_TIMEOUT** (p. 316)).

If there are no instance resources left, this operation may fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315). Calling **FooDataWriter::unregister_instance** (p. 1480) may help freeing up some resources.

This operation will fail with **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) if the timestamp is less than the timestamp used in the last writer operation (*register*, *unregister*, *dispose*, or *write*, with either the automatically supplied timestamp or the application-provided timestamp).

Parameters:

instance_data *<<in>> (p. 200)* The data to write.

Parameters:

handle *<<in>> (p. 200)* Either the handle returned by a previous call to **FooDataWriter::register_instance** (p. 1478), or else the special value **DDS_HANDLE_NIL** (p. 55). If **Foo** (p. 1443) *has* a key and `handle` is not **DDS_HANDLE_NIL** (p. 55), `handle` must represent a registered instance of type **Foo** (p. 1443). Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315), or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

DDSDataReader (p. 1087)
FooDataWriter::write_w_timestamp (p. 1486)
DESTINATION_ORDER (p. 365)

6.233.2.9 virtual DDS_ReturnCode_t FooDataWriter::write_w_timestamp (const Foo & *instance_data*, const DDS_InstanceHandle_t & *handle*, const DDS_Time_t & *source_timestamp*) [pure virtual]

Performs the same function as **FooDataWriter::write** (p. 1484) except that it also provides the value for the `source_timestamp`.

Explicitly provides the timestamp that will be available to the **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912). (Refer to **DDS_SampleInfo** (p. 912) and **DESTINATION_ORDER** (p. 365) QoS policy for details)

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the **FooDataWriter::write** (p. 1484) operation.

This operation may block and time out (**DDS_RETCODE_TIMEOUT** (p. 316)) under the same circumtances described for **FooDataWriter::write** (p. 1484).

If there are no instance resources left, this operation may fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315). Calling **FooDataWriter::unregister_instance** (p. 1480) may help free up some resources.

This operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) under the same circumstances described for the write operation.

Parameters:

instance_data *<<in>> (p. 200)* The data to write.

handle *<<in>> (p. 200)* Either the handle returned by a previous call to **FooDataWriter::register_instance** (p. 1478), or else the special value **DDS_HANDLE_NIL** (p. 55). If **Foo** (p. 1443) *has* a key and `handle` is not **DDS_HANDLE_NIL** (p. 55), `handle` must represent

a registered instance of type **Foo** (p. 1443). Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

source_timestamp *<<in>>* *(p. 200)* When using **DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS** (p. 366) the timestamp value must be greater than or equal to the timestamp value used in the last writer operation (*register*, *unregister*, *dispose*, or *write*, with either the automatically supplied timestamp or the application-provided timestamp) However, if it is less than the timestamp of the previous operation but the difference is less than the **DDS_DestinationOrderQosPolicy::source_timestamp_tolerance** (p. 572), the timestamp of the previous operation will be used as the source timestamp of this sample. Otherwise, if the difference is greater than **DDS_DestinationOrderQosPolicy::source_timestamp_tolerance** (p. 572), the function will return **DDS_RETCODE_BAD_PARAMETER** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315), or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataWriter::write (p. 1484)
DDSDataReader (p. 1087)
DESTINATION_ORDER (p. 365)

6.233.2.10 virtual DDS_ReturnCode_t FooDataWriter::write_w_params (const Foo & *instance_data*, DDS_WriteParams_t & *params*) [pure virtual]

Performs the same function as **FooDataWriter::write** (p. 1484) and **FooDataWriter::write_w_timestamp** (p. 1486) except that it also provides the values contained in `params`.

Allows provision of the sample identity, instance handle, source timestamp, publication priority, and cookie contained in `params`.

The sample identity identifies the sample being written. The identity consist of a pair (virtual GUID, virtual sequence number).

The cookie is a sequence of bytes tagging the data being written, and is used by the callback DDSDataWriterListener::on_application_acknowledgment.

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the **FooDataWriter::write** (p. 1484) operation.

This operation may block and time out (**DDS_RETCODE_TIMEOUT** (p. 316)) under the same circumtances described for **FooDataWriter::write** (p. 1484).

If there are no instance resources left, this operation may fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315). Calling **FooDataWriter::unregister_instance_w_params** (p. 1483) may help free up some resources.

This operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) under the same circumstances described for the write operation.

Parameters:

instance_data *<<in>>* *(p. 200)* The data to write.

params *<<in>>* *(p. 200)*

The handle is either returned by a previous call to **FooDataWriter::register_instance** (p. 1478), or else the special value **DDS_HANDLE_NIL** (p. 55). If **Foo** (p. 1443) *has* a key and `handle` is not **DDS_HANDLE_NIL** (p. 55), `handle` must represent a registered instance of type **Foo** (p. 1443). Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

The source_timestamp value must be greater than or equal to the timestamp value used in the last writer operation (used in a *register*, *unregister*, *dispose*, or *write*, with either the automatically supplied timestamp or the application provided timestamp). This timestamp may potentially affect the order in which readers observe events from multiple writers. This timestamp will be available to the **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataWriter::write (p. 1484)
DDSDataReader (p. 1087)
DESTINATION_ORDER (p. 365)

6.233.2.11 virtual DDS_ReturnCode_t FooDataWriter::dispose (const Foo & *instance_data*, const DDS_InstanceHandle_t & *instance_handle*) [pure virtual]

Requests the middleware to delete the data.

This operation is useful only for keyed data types. Using it for non-keyed types has no effect and reports no error.

The actual deletion is postponed until there is no more use for that data in the whole system.

Applications are made aware of the deletion by means of operations on the **DDSDataReader** (p. 1087) objects that already knew that instance. **DDSDataReader** (p. 1087) objects that didn't know the instance will never see it.

This operation does not modify the value of the instance. The `instance_data` parameter is passed just for the purposes of identifying the instance.

When this operation is used, RTI Connext will automatically supply the value of the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

The constraints on the values of the handle parameter and the corresponding error behavior are the same specified for the **FooDataWriter::unregister_instance** (p. 1480) operation.

The special value **DDS_HANDLE_NIL** (p. 55) can be used for the parameter `instance_handle`. This indicates the identity of the instance should be automatically deduced from the `instance_data` (by means of the `key`).

If `handle` is any value other than **DDS_HANDLE_NIL** (p. 55), then it must correspond to an instance that has been registered. If there is no correspondence, the operation will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

RTI Connext will not detect the error when the `handle` is any value other than **DDS_HANDLE_NIL** (p. 55), corresponds to an instance that has been registered, but does not correspond to the instance deduced from the `instance_data` (by means of the `key`). RTI Connext will treat as if the **dispose()** (p. 1489) operation is for the instance as indicated by the `handle`.

This operation may block and time out (**DDS_RETCODE_TIMEOUT** (p. 316)) under the same circumtances described for **FooDataWriter::write()** (p. 1484).

If there are no instance resources left, this operation may fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315). Calling **FooDataWriter::unregister_instance** (p. 1480) may help freeing up some re-

sources.

Parameters:

instance_data *<<in>>* *(p. 200)* The data to dispose. If **Foo** (p. 1443) *has* a key and `instance_handle` is **DDS_HANDLE_NIL** (p. 55), only the fields that represent the key are examined by the function. Otherwise, `instance_data` is not used.

instance_handle *<<in>>* *(p. 200)* Either the handle returned by a previous call to **FooDataWriter::register_instance** (p. 1478), or else the special value **DDS_HANDLE_NIL** (p. 55). If **Foo** (p. 1443) *has* a key and `instance_handle` is **DDS_HANDLE_NIL** (p. 55), `instance_handle` is not used and `instance` is deduced from `instance_data`. If **Foo** (p. 1443) has no key, `instance_handle` is not used. If `handle` is used, it must represent a registered instance of type **Foo** (p. 1443). Otherwise, this method fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataWriter::dispose_w_timestamp (p. 1490)
RELATIONSHIP BETWEEN REGISTRATION, LIVELINESS and OWNERSHIP (p. 809)

6.233.2.12 virtual DDS_ReturnCode_t FooDataWriter::dispose_w_timestamp (const Foo & *instance_data*, const DDS_InstanceHandle_t & *instance_handle*, const DDS_Time_t & *source_timestamp*) [pure virtual]

Performs the same functions as dispose except that the application provides the value for the `source_timestamp` that is made available to **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the **FooDataWriter::dispose** (p. 1489) operation.

This operation may block and time out (**DDS_RETCODE_TIMEOUT** (p. 316)) under the same circumtances described for **FooDataWriter::write** (p. 1484).

If there are no instance resources left, this operation may fail with **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315). Calling **FooDataWriter::unregister_instance** (p. 1480) may help freeing up some resources.

Parameters:

instance_data *<<in>> (p. 200)* The data to dispose. If **Foo** (p. 1443) *has* a key and `instance_handle` is **DDS_HANDLE_NIL** (p. 55), only the fields that represent the key are examined by the function. Otherwise, `instance_data` is not used.

instance_handle *<<in>> (p. 200)* Either the handle returned by a previous call to **FooDataWriter::register_instance** (p. 1478), or else the special value **DDS_HANDLE_NIL** (p. 55). If **Foo** (p. 1443) *has* a key and `instance_handle` is **DDS_HANDLE_NIL** (p. 55), `instance_handle` is not used and `instance` is deduced from `instance_data`. If **Foo** (p. 1443) has no key, `instance_handle` is not used. If `handle` is used, it must represent a registered instance of type **Foo** (p. 1443). Otherwise, this method may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315)

source_timestamp *<<in>> (p. 200)* The timestamp value must be greater than or equal to the timestamp value used in the last writer operation (used in a *register*, *unregister*, *dispose*, or *write*, with either the automatically supplied timestamp or the application provided timestamp). This timestamp may potentially affect the order in which readers observe events from multiple writers. This timestamp will be available to the **DDSDataReader** (p. 1087) objects by means of the `source_timestamp` attribute inside the **DDS_SampleInfo** (p. 912).

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_TIMEOUT** (p. 316), **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataWriter::dispose (p. 1489)

6.233.2.13 virtual DDS_ReturnCode_t FooDataWriter::dispose_w_params (const Foo & *instance_data*, DDS_WriteParams_t & *params*) [pure virtual]

Performs the same function as **FooDataWriter::dispose** (p. 1489) and **FooDataWriter::dispose_w_timestamp** (p. 1490) except that it also provides the values contained in `params`.

See also:

FooDataWriter::write_w_params (p. 1487)

6.233.2.14 virtual DDS_ReturnCode_t FooDataWriter::get_key_value (Foo & *key_holder*, const DDS_InstanceHandle_t & *handle*) [pure virtual]

Retrieve the instance `key` that corresponds to an instance `handle`.

Useful for keyed data types.

The operation will only fill the fields that form the `key` inside the `key_holder` instance. If **Foo** (p. 1443) has no key, this method has no effect and exit with no error.

For keyed data types, this operation may fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315) if the `handle` does not correspond to an existing data-object known to the **DDSDataWriter** (p. 1113).

Parameters:

key_holder *<<inout>> (p. 200)* a user data type specific key holder, whose `key` fields are filled by this operation. If **Foo** (p. 1443) has no key, this method has no effect.

handle *<<in>> (p. 200)* the `instance` whose key is to be retrieved. If **Foo** (p. 1443) *has* a key, `handle` must represent a registered instance of type **Foo** (p. 1443). Otherwise, this method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315). If **Foo** (p. 1443) *has* a key and `handle` is **DDS_HANDLE_NIL** (p. 55), this method will fail with **DDS_RETCODE_BAD_PARAMETER** (p. 315).

Returns:

One of the **Standard Return Codes** (p. 314) or **DDS_RETCODE_NOT_ENABLED** (p. 315).

See also:

FooDataReader::get_key_value (p. 1472)

6.233.2.15 virtual DDS_InstanceHandle_t FooDataWriter::lookup_instance (const Foo & *key_holder*) [pure virtual]

Retrieve the instance `handle` that corresponds to an instance `key_holder`.

Useful for keyed data types.

This operation takes as a parameter an instance and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The instance parameter is only used for the purpose of examining the fields that define the key. This operation does not register the instance in question. If the instance has not been previously registered, or if for any other reason RTI Connext is unable to provide an instance handle, RTI Connext will return the special value HANDLE_NIL.

Parameters:

key_holder *<<in>>* *(*p. 200) a user data type specific key holder.

Returns:

the instance handle associated with this instance. If **Foo** (p. 1443) has no key, this method has no effect and returns **DDS_HANDLE_NIL** (p. 55)

6.234 FooSeq Struct Reference

<<***interface***>> *(p. 199)* <<***generic***>> *(p. 199)* A type-safe, ordered collection of elements. The type of these elements is referred to in this documentation as `Foo` (p. 1443).

Public Member Functions

- **FooSeq** & **operator=** (const struct **FooSeq** &src_seq)

  Copy elements from another sequence, resizing the sequence if necessary.

- bool **copy_no_alloc** (const struct **FooSeq** &src_seq)

  Copy elements from another sequence, only if the destination sequence has enough capacity.

- bool **from_array** (const **Foo** array[], **DDS_Long** length)

  Copy elements from an array of elements, resizing the sequence if necessary. The original contents of the sequence (if any) are replaced.

- bool **to_array** (**Foo** array[], **DDS_Long** length)

  Copy elements to an array of elements. The original contents of the array (if any) are replaced.

- **Foo** & **operator[]** (**DDS_Long** i)

  Set the `i-th` *element of the sequence.*

- const **Foo** & **operator[]** (**DDS_Long** i) const

  Get the `i-th` *element for a* `const` *sequence.*

- **DDS_Long length** () const

  Get the logical length of this sequence.

- bool **length** (**DDS_Long** new_length)

  Change the length of this sequence.

- bool **ensure_length** (**DDS_Long** length, **DDS_Long** max)

  Set the sequence to the desired length, and resize the sequence if necessary.

- **DDS_Long maximum** () const

  Get the current maximum number of elements that can be stored in this sequence.

- bool **maximum** (**DDS_Long** new_max)

Resize this sequence to a new desired maximum.

- bool **loan_contiguous** (**Foo** *buffer, **DDS_Long** new_length, **DDS_Long** new_max)

  Loan a contiguous buffer to this sequence.

- bool **loan_discontiguous** (**Foo** **buffer, **DDS_Long** new_length, **DDS_Long** new_max)

  Loan a discontiguous buffer to this sequence.

- bool **unloan** ()

  Return the loaned buffer in the sequence and set the maximum to 0.

- **Foo** * **get_contiguous_buffer** () const

  Return the contiguous buffer of the sequence.

- **Foo** ** **get_discontiguous_buffer** () const

  Return the discontiguous buffer of the sequence.

- bool **has_ownership** ()

  Return the value of the owned flag.

- ~**FooSeq** ()

  Deallocate this sequence's buffer.

- **FooSeq** (**DDS_Long** new_max=0)

  Create a sequence with the given maximum.

- **FooSeq** (const struct **FooSeq** &foo_seq)

  Create a sequence by copying from an existing sequence.

6.234.1 Detailed Description

<<***interface***>> *(p. 199)* <<***generic***>> *(p. 199)* A type-safe, ordered collection of elements. The type of these elements is referred to in this documentation as `Foo (p. 1443)`.

For users who define data types in OMG IDL, this type corresponds to the IDL express `sequence<Foo (p. 1443)>`.

For any user-data type `Foo (p. 1443)` that an application defines for the purpose of data-distribution with RTI Connext, a `FooSeq (p. 1494)` is generated. The sequence offers a subset of the methods defined by the standard OMG IDL to

C++ mapping for sequences. We refer to an IDL `sequence<Foo (p. 1443)>` as `FooSeq (p. 1494)`.

The state of a sequence is described by the properties 'maximum', 'length' and 'owned'.

- The 'maximum' represents the size of the underlying buffer; this is the maximum number of elements it can possibly hold. It is returned by the **FooSeq::maximum()** (p. 1503) operation.

- The 'length' represents the actual number of elements it currently holds. It is returned by the **FooSeq::length()** (p. 1501) operation.

- The 'owned' flag represents whether the sequence owns the underlying buffer. It is returned by the **FooSeq::has_ownership** (p. 1508) operation. If the sequence does not own the underlying buffer, the underlying buffer is loaned from somewhere else. This flag influences the lifecycle of the sequence and what operations are allowed on it. The general guidelines are provided below and more details are described in detail as pre-conditions and post-conditions of each of the sequence's operations:
  - If owned == **DDS_BOOLEAN_TRUE** (p. 298), the sequence has ownership on the buffer. It is then responsible for destroying the buffer when the sequence is destroyed.
  - If the owned == **DDS_BOOLEAN_FALSE** (p. 299), the sequence does not have ownership on the buffer. This implies that the sequence is loaning the buffer. The sequence cannot be destroyed until the loan is returned.
  - A sequence with a zero maximum always has owned == **DDS_BOOLEAN_TRUE** (p. 298)

See also:

FooDataWriter (p. 1475), **FooDataReader** (p. 1444), **FooTypeSupport** (p. 1509), **rtiddsgen** (p. 220)

6.234.2 Constructor & Destructor Documentation

6.234.2.1 FooSeq::~FooSeq ()

Deallocate this sequence's buffer.

Precondition:

(owned == **DDS_BOOLEAN_TRUE** (p. 298)). If this precondition is not met, no memory will be freed and an error will be logged.

Postcondition:

maximum == 0 and the underlying buffer is freed.

See also:

FooSeq::maximum() (p. 1503), **FooSeq::unloan** (p. 1506)

6.234.2.2 FooSeq::FooSeq (DDS_Long *new_max* = 0)

Create a sequence with the given maximum.

This is a constructor for the sequence. The constructor will automatically allocate memory to hold new_max elements of type **Foo** (p. 1443).

This constructor will be used when the application creates a sequence using one of the following:

```
FooSeq mySeq(5);
 // or
 FooSeq mySeq;
 // or
 FooSeq* mySeqPtr = new FooSeq(5);
```

Postcondition:

maximum == new_max
length == 0
owned == **DDS_BOOLEAN_TRUE** (p. 298),

Parameters:

new_max Must be >= 0. Otherwise the sequence will be initialized to a new_max=0.

6.234.2.3 FooSeq::FooSeq (const struct FooSeq & *foo_seq*)

Create a sequence by copying from an existing sequence.

This is a constructor for the sequence. The constructor will automatically allocate memory to hold foo_seq::maximum() elements of type **Foo** (p. 1443) and will copy the current contents of foo_seq into the new sequence.

This constructor will be used when the application creates a sequence using one of the following:

```
FooSeq mySeq(foo_seq);
 // or
```

```
FooSeq mySeq = foo_seq;
// or
FooSeq *mySeqPtr = new FooSeq(foo_seq);
```

Postcondition:

this::maximum == foo_seq::maximum
this::length == foo_seq::length
this[i] == foo_seq[i] for 0 <= i < foo_seq::length
this::owned == **DDS_BOOLEAN_TRUE** (p. 298)

Note:

If the pre-conditions are not met, the constructor will initialize the new sequence to a maximum of zero.

6.234.3 Member Function Documentation

6.234.3.1 FooSeq& FooSeq::operator= (const struct FooSeq & *src_seq*)

Copy elements from another sequence, resizing the sequence if necessary.

This method invokes **FooSeq::copy_no_alloc** (p. 1499) after ensuring that the sequence has enough capacity to hold the elements to be copied.

This operator is invoked when the following expression appears in the code:

```
target_seq = src_seq
```

Important: This method *will* allocate memory if this::maximum < src_seq::length.

Therefore, to programatically detect the successful completion of the operator it is recommended that the application first sets the length of this sequence to zero, makes the assignment, and then checks that the length of this sequence matches that of src_seq.

Parameters:

src_seq *<<in>> (p. 200)* the sequence from which to copy

See also:

FooSeq::copy_no_alloc (p. 1499)

6.234.3.2 bool FooSeq::copy_no_alloc (const struct FooSeq & *src_seq*)

Copy elements from another sequence, only if the destination sequence has enough capacity.

Fill the elements in this sequence by copying the corresponding elements in `src_seq`. The original contents in this sequence are replaced via the element assignment operation (Foo_copy() function). By default, elements are discarded; 'delete' is not invoked on the discarded elements.

Precondition:

this::maximum >= src_seq::length
this::owned == **DDS_BOOLEAN_TRUE** (p. 298)

Postcondition:

this::length == src_seq::length
this[i] == src_seq[i] for 0 <= i < target_seq::length
this::owned == **DDS_BOOLEAN_TRUE** (p. 298)

Parameters:

src_seq <<***in***>> *(p. 200)* the sequence from which to copy

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the sequence was successfully copied; **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

Note:

If the pre-conditions are not met, the operator will print a message to stdout and leave this sequence unchanged.

See also:

FooSeq::operator= (p. 1498)

6.234.3.3 bool FooSeq::from_array (const Foo *array*[], DDS_Long *length*)

Copy elements from an array of elements, resizing the sequence if necessary. The original contents of the sequence (if any) are replaced.

Fill the elements in this sequence by copying the corresponding elements in `array`. The original contents in this sequence are replaced via the element assignment operation (Foo_copy() function). By default, elements are discarded; 'delete' is not invoked on the discarded elements.

Precondition:

this::owned == **DDS_BOOLEAN_TRUE** (p. 298)

Postcondition:

this::length == `length`
this[i] == array[i] for 0 <= i < `length`
this::owned == **DDS_BOOLEAN_TRUE** (p. 298)

Parameters:

array *<<**in**>> (p. 200)* The array of elements to be copy elements from

length *<<**in**>> (p. 200)* The length of the array.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the array was successfully copied; **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

Note:

If the pre-conditions are not met, the method will print a message to stdout and leave this sequence unchanged.

6.234.3.4 bool FooSeq::to_array (Foo *array*[], DDS_Long *length*)

Copy elements to an array of elements. The original contents of the array (if any) are replaced.

Copy the elements of this sequence to the corresponding elements in the array. The original contents of the array are replaced via the element assignment operation (Foo_copy() function). By default, elements are discarded; 'delete' is not invoked on the discarded elements.

Parameters:

array *<<**in**>> (p. 200)* The array of elements to be filled with elements from this sequence

length *<<**in**>> (p. 200)* The number of elements to be copied.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the elements of the sequence were successfully copied; **DDS_BOOLEAN_FALSE** (p. 299) otherwise.

6.234.3.5 Foo& FooSeq::operator[] (DDS_Long *i*)

Set the `i-th` element of the sequence.

This is the operator that is invoked when the application indexes into a non-`const` sequence:

```
myElement = mySequence[i];
 mySequence[i] = myElement;
```

Note that a *reference* to the `i-th` element is returned (and not a copy).

Parameters:

i index of element to access, must be >= 0 and less than **FooSeq::length()** (p. 1501)

Returns:

the `i-th` element

6.234.3.6 const Foo& FooSeq::operator[] (DDS_Long *i*) const

Get the `i-th` element for a `const` sequence.

This is the operator that is invoked when the application indexes into a `const` sequence:

```
myElement = mySequence[i];
```

Note that a *reference* to the `i-th` element is returned (and not a copy).

Parameters:

i index of element to access, must be >= 0 and less than **FooSeq::length()** (p. 1501)

Returns:

the `i-th` element

6.234.3.7 DDS_Long FooSeq::length () const

Get the logical length of this sequence.

Get the length that was last set, or zero if the length has never been set.

Returns:

the length of the sequence

6.234.3.8 bool FooSeq::length (DDS_Long *new_length*)

Change the length of this sequence.

This method does not allocate/deallocate memory.

The new length must not exceed the maximum of this sequence as returned by the **FooSeq::maximum()** (p. 1503) operation. (Note that, if necessary, the maximum of this sequence can be increased manually by using the FooSeq::maximum(long) operation.)

The elements of the sequence are not modified by this operation. If the new length is larger than the original length, the new elements will be uninitialized; if the length is decreased, the old elements that are beyond the new length will physically remain in the sequence but will not be accessible.

Postcondition:

length = new_length.

Parameters:

new_length the new desired length. This value must be non-negative and cannot exceed maximum of the sequence. In other words 0 <= new_length <= maximum

Returns:

DDS_BOOLEAN_TRUE (p. 298) on sucess or **DDS_BOOLEAN_FALSE** (p. 299) on failure

6.234.3.9 bool FooSeq::ensure_length (DDS_Long *length*, DDS_Long *max*)

Set the sequence to the desired length, and resize the sequence if necessary.

If the current maximum is greater than the desired length, then sequence is not resized.

Otherwise if this sequence owns its buffer, the sequence is resized to the new maximum by freeing and re-allocating the buffer. However, if the sequence does not own its buffer, this operation will fail.

This function allows user to avoid unnecessary buffer re-allocation.

Precondition:

`length` <= `max`
owned == **DDS_BOOLEAN_TRUE** (p. 298) if sequence needs to be resized

Postcondition:

length == `length`
maximum == `max` if resized

Parameters:

length *<<in>>* *(p. 200)* The new length that should be set. Must be >= 0.

max *<<in>>* *(p. 200)* If sequence need to be resized, this is the maximum that should be set. `max` >= `length`

Returns:

DDS_BOOLEAN_TRUE (p. 298) on success, **DDS_BOOLEAN_FALSE** (p. 299) if the preconditions are not met. In that case the sequence is not modified.

6.234.3.10 DDS_Long FooSeq::maximum () const

Get the current maximum number of elements that can be stored in this sequence.

The `maximum` of the sequence represents the maximum number of elements that the underlying buffer can hold. It does not represent the current number of elements.

The `maximum` is a non-negative number. It is initialized when the sequence is first created.

`maximum` can only be changed with the FooSeq::maximum(long) operation.

Returns:

the current maximum of the sequence.

See also:

FooSeq::length() (p. 1501)

6.234.3.11 bool FooSeq::maximum (DDS_Long *new_max*)

Resize this sequence to a new desired maximum.

This operation does nothing if the new desired maximum matches the current maximum.

If this sequence owns its buffer and the new maximum is not equal to the old maximum, then the existing buffer will be freed and re-allocated.

Precondition:

owned == **DDS_BOOLEAN_TRUE** (p. 298)

Postcondition:

owned == **DDS_BOOLEAN_TRUE** (p. 298)
length == MINIMUM(original length, new_max)

Parameters:

new_max Must be >= 0.

Returns:

DDS_BOOLEAN_TRUE (p. 298) on success, **DDS_BOOLEAN_FALSE** (p. 299) if the preconditions are not met. In that case the sequence is not modified.

6.234.3.12 bool FooSeq::loan_contiguous (Foo * *buffer*, DDS_Long *new_length*, DDS_Long *new_max*)

Loan a contiguous buffer to this sequence.

This operation changes the `owned` flag of the sequence to **DDS_BOOLEAN_FALSE** (p. 299) and also sets the underlying buffer used by the sequence. See the `User's Manual` for more information about sequences and memory ownership.

Use this method if you want to manage the memory used by the sequence yourself. You must provide an array of elements and integers indicating how many elements are allocated in that array (i.e. the maximum) and how many elements are valid (i.e. the length). The sequence will subsequently use the memory you provide and will not permit it to be freed by a call to FooSeq::maximum(long).

Once you have loaned a buffer to a sequence, make sure that you don't free it before calling **FooSeq::unloan** (p. 1506): the next time you access the sequence, you will be accessing freed memory!

You can use this method to wrap stack memory with a sequence interface, thereby avoiding dynamic memory allocation. Create a **FooSeq** (p. 1494) and an array of type **Foo** (p. 1443) and then loan the array to the sequence:

```
::Foo fooArray[10];
 ::FooSeq fooSeq;
fooSeq.loan_contiguous(fooArray, 0, 10);
```

By default, a sequence you create owns its memory unless you explicitly loan memory of your own to it. In a very few cases, RTI Connext will return a sequence to you that has a loan; those cases are documented as such. For example, if you call **FooDataReader::read** (p. 1447) or **FooDataReader::take** (p. 1448) and pass in sequences with no loan and no memory allocated, RTI Connext will loan memory to your sequences which must be unloaned with **FooDataReader::return_loan** (p. 1471). See the documentation of those methods for more information.

Precondition:

FooSeq::maximum() (p. 1503) == 0; i.e. the sequence has no memory allocated to it.
FooSeq::has_ownership (p. 1508) == **DDS_BOOLEAN_TRUE** (p. 298); i.e. the sequence does not already have an outstanding loan

Postcondition:

The sequence will store its elements in the buffer provided.
FooSeq::has_ownership (p. 1508) == **DDS_BOOLEAN_FALSE** (p. 299)
FooSeq::length() (p. 1501) == new_length
FooSeq::maximum() (p. 1503) == new_max

Parameters:

buffer The new buffer that the sequence will use. Must point to enough memory to hold new_max elements of type **Foo** (p. 1443). It may be NULL if new_max == 0.

new_length The desired new length for the sequence. It must be the case that that 0 <= new_length <= new_max.

new_max The allocated number of elements that could fit in the loaned buffer.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if `buffer` is successfully loaned to this sequence or **DDS_BOOLEAN_FALSE** (p. 299) otherwise. Failure only occurs due to failing to meet the pre-conditions. Upon failure the sequence remains unmodified.

See also:

FooSeq::unloan (p. 1506), **FooSeq::loan_discontiguous** (p. 1506)

6.234.3.13 bool FooSeq::loan_discontiguous (Foo ** *buffer*, DDS_Long *new_length*, DDS_Long *new_max*)

Loan a discontiguous buffer to this sequence.

This method is exactly like **FooSeq::loan_contiguous** (p. 1504) except that the buffer loaned is an array of **Foo** (p. 1443) pointers, not an array of **Foo** (p. 1443).

Parameters:

buffer The new buffer that the sequence will use. Must point to enough memory to hold new_max elements of type Foo*. It may be NULL if new_max == 0.

new_length The desired new length for the sequence. It must be the case that that 0 <= new_length <= new_max.

new_max The allocated number of elements that could fit in the loaned buffer.

See also:

FooSeq::unloan (p. 1506), **FooSeq::loan_contiguous** (p. 1504)

6.234.3.14 bool FooSeq::unloan ()

Return the loaned buffer in the sequence and set the maximum to 0.

This method affects only the state of this sequence; it does not change the contents of the buffer in any way.

Only the user who originally loaned a buffer should return that loan, as the user may have dependencies on that memory known only to them. Unloaning someone else's buffer may cause unspecified problems. For example, suppose a sequence is loaning memory from a custom memory pool. A user of the sequence likely has no way to release the memory back into the pool, so unloaning the sequence buffer would result in a resource leak. If the user were to then re-loan a different buffer, the original creator of the sequence would have no way to discover, when freeing the sequence, that the loan no longer referred to its own memory and would thus not free the user's memory properly, exacerbating the situation and leading to undefined behavior.

Precondition:

owned == **DDS_BOOLEAN_FALSE** (p. 299)

Postcondition:

owned == **DDS_BOOLEAN_TRUE** (p. 298)
maximum == 0

Returns:

DDS_BOOLEAN_TRUE (p. 298) if the preconditions were met. Otherwise **DDS_BOOLEAN_FALSE** (p. 299). The function only fails if the pre-conditions are not met, in which case it leaves the sequence unmodified.

See also:

FooSeq::loan_contiguous (p. 1504), **FooSeq::loan_discontiguous** (p. 1506), FooSeq::maximum(long)

6.234.3.15 Foo* FooSeq::get_contiguous_buffer () const

Return the contiguous buffer of the sequence.

Get the underlying buffer where contiguous elements of the sequence are stored. The size of the buffer matches the maximum of the sequence, but only the elements up to the **FooSeq::length()** (p. 1501) of the sequence are valid.

This method provides almost no encapsulation of the sequence's underlying implementation. Certain operations, such as FooSeq::maximum(long), may render the buffer invalid. In light of these caveats, this operation should be used with care.

Returns:

buffer that stores contiguous elements in sequence.

6.234.3.16 Foo** FooSeq::get_discontiguous_buffer () const

Return the discontiguous buffer of the sequence.

This operation returns the underlying buffer where discontiguous elements of the sequence are stored. The size of the buffer matches the maximum of this sequence, but only the elements up to the **FooSeq::length()** (p. 1501) of the sequence are valid.

The same caveats apply to this method as to FooSeq::get_contiguous_bufferI.

The sequence will dereference pointers in the discontiguous buffer to provide access to its elements by value in C and by reference in C++. If you access the discontiguous buffer directly by means of this method, do not store any NULL values into it, as acessing those values will result in a segmentation fault.

Returns:

buffer that stores discontiguous elements in sequence.

6.234.3.17 bool FooSeq::has_ownership ()

Return the value of the owned flag.

Returns:

DDS_BOOLEAN_TRUE (p. 298) if sequence owns the underlying buffer, or **DDS_BOOLEAN_FALSE** (p. 299) if it has an outstanding loan.

6.235 FooTypeSupport Struct Reference

*<<**interface**>> (p. 199) <<**generic**>> (p. 199)* User data type specific interface.

Inheritance diagram for FooTypeSupport::

Static Public Member Functions

- static **DDS_ReturnCode_t register_type** (**DDSDomainParticipant** *participant, const char *type_name)

  Allows an application to communicate to RTI Connext the existence of a data type.

- static **DDS_ReturnCode_t unregister_type** (**DDSDomainParticipant** *participant, const char *type_name)

  Allows an application to unregister a data type from RTI Connext. After calling unregister_type, no further communication using that type is possible.

- static **Foo** * **create_data** ()

  <<**eXtension**>> (p. 199) *Create a data type and initialize it.*

- static **Foo** * **create_data_ex** (**DDS_Boolean** allocatePointers)

  <<**eXtension**>> (p. 199) *Create a data type and initialize it.*

- static **DDS_ReturnCode_t copy_data** (**Foo** *dst_data, const **Foo** *src_data)

  <<**eXtension**>> (p. 199) *Copy data type.*

- static **DDS_ReturnCode_t delete_data** (**Foo** *a_data)

  <<**eXtension**>> (p. 199) *Destroy a user data type instance.*

- static **DDS_ReturnCode_t delete_data_ex** (**Foo** *a_data, **DDS_Boolean** deletePointers)

  <<**eXtension**>> (p. 199) *Destroy a user data type instance.*

- static **DDS_ReturnCode_t initialize_data** (**Foo** *a_data)

  <<**eXtension**>> (p. 199) *Initialize data type.*

- static **DDS_ReturnCode_t initialize_data_ex** (**Foo** *a_data, **DDS_Boolean** allocatePointers)

<<**eXtension**>> (p. 199) *Initialize data type.*

- static **DDS_ReturnCode_t finalize_data** (**Foo** *a_data)

  <<**eXtension**>> (p. 199) *Finalize data type.*

- static **DDS_ReturnCode_t finalize_data_ex** (**Foo** *a_data, **DDS_Boolean** deletePointers)

  <<**eXtension**>> (p. 199) *Finalize data type.*

- static const char * **get_type_name** ()

  Get the default name for this type.

- static void **print_data** (const **Foo** *a_data)

  <<**eXtension**>> (p. 199) *Print value of data type to standard out.*

6.235.1 Detailed Description

<<***interface***>> *(p. 199)* <<***generic***>> *(p. 199)* User data type specific interface.

Defines the user data type specific interface generated for each application class.

The concrete user data type automatically generated by the implementation is an incarnation of this class.

See also:

DDS_TYPESUPPORT_CPP (p. 52)
rtiddsgen (p. 220)

6.235.2 Member Function Documentation

6.235.2.1 static DDS_ReturnCode_t FooTypeSupport::register_type (DDSDomainParticipant * *participant*, const char * *type_name*) [static]

Allows an application to communicate to RTI Connext the existence of a data type.

The *generated* implementation of the operation embeds all the knowledge that has to be communicated to the middleware in order to make it able to manage the contents of data of that type. This includes in particular the key definition that will allow RTI Connext to distinguish different instances of the same type.

The same **DDSTypeSupport** (p. 1432) can be registered multiple times with a **DDSDomainParticipant** (p. 1139) using the same or different values for the `type_name`. If `register_type` is called multiple times on the same **DDSTypeSupport** (p. 1432) with the same **DDSDomainParticipant** (p. 1139) and `type_name`, the second (and subsequent) registrations are ignored by the operation fails with **DDS_RETCODE_OK** (p. 315).

Precondition:

Cannot use the same `type_name` to register two different **DDSTypeSupport** (p. 1432) with the same **DDSDomainParticipant** (p. 1139), or else the operation will fail and **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) will be returned.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to register the data type `Foo` (p. 1443) with. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type `Foo` (p. 1443) is registered with the participant; this type name is used when creating a new **DDSTopic** (p. 1419). (See **DDSDomainParticipant::create_topic** (p. 1175).) The name may not be NULL or longer than 255 characters.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315) or **DDS_RETCODE_OUT_OF_RESOURCES** (p. 315).

MT Safety:

UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to register a type. Subsequent calls are thread safe.

See also:

DDSDomainParticipant::create_topic (p. 1175)

6.235.2.2 static DDS_ReturnCode_t FooTypeSupport::unregister_type (DDSDomainParticipant * *participant*, const char * *type_name*) [static]

Allows an application to unregister a data type from RTI Connext. After calling unregister_type, no further communication using that type is possible.

The *generated* implementation of the operation removes all the information about a type from RTI Connext. No further communication using that type is possible.

Precondition:

A type with `type_name` is registered with the participant and all **DDSTopic** (p. 1419) objects referencing the type have been destroyed. If the type is not registered with the participant, or if any **DDSTopic** (p. 1419) is associated with the type, the operation will fail with **DDS_RETCODE_ERROR** (p. 315).

Postcondition:

All information about the type is removed from RTI Connext. No further communication using this type is possible.

Parameters:

participant *<<in>> (p. 200)* the **DDSDomainParticipant** (p. 1139) to unregister the data type `Foo` (p. 1443) from. Cannot be NULL.

type_name *<<in>> (p. 200)* the type name under with the data type `Foo` (p. 1443) is registered with the participant. The name should match a name that has been previously used to register a type with the participant. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314), **DDS_RETCODE_BAD_PARAMETER** (p. 315) or **DDS_RETCODE_ERROR** (p. 315)

MT Safety:

SAFE.

See also:

FooTypeSupport::register_type (p. 1510)

6.235.2.3 static Foo* FooTypeSupport::create_data () [static]

<<eXtension>> (p. 199) Create a data type and initialize it.

The *generated* implementation of the operation knows how to instantiate a data type and initialize it properly.

All memory for the type is deeply allocated.

Returns:

newly created data type

See also:

FooTypeSupport::delete_data (p. 1514)

6.235.2.4 static Foo* FooTypeSupport::create_data_ex (DDS_Boolean *allocatePointers*) [static]

<<eXtension>> (p. 199) Create a data type and initialize it.

The *generated* implementation of the operation knows how to instantiate a data type and initialize it properly.

When `allocatePointers` is **DDS_BOOLEAN_TRUE** (p. 298), all the references (pointers) in the type are recursively allocated.

Parameters:

allocatePointers *<<in>>* (p. 200) Whether or not to recursively allocate pointers.

Returns:

newly created data type

See also:

FooTypeSupport::delete_data_ex (p. 1514)

6.235.2.5 static DDS_ReturnCode_t FooTypeSupport::copy_data (Foo * *dst_data*, const Foo * *src_data*) [static]

<<eXtension>> (p. 199) Copy data type.

The *generated* implementation of the operation knows how to copy value of a data type.

Parameters:

dst_data *<<inout>>* (p. 200) Data type to copy value to. Cannot be NULL.

src_data *<<in>>* (p. 200) Data type to copy value from. Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314)

6.235.2.6 static DDS_ReturnCode_t FooTypeSupport::delete_data (Foo * *a_data*) [static]

*<<**eXtension**>>* *(*p. *199)* Destroy a user data type instance.

The *generated* implementation of the operation knows how to destroy a data type and return all resources.

Parameters:

a_data *<<**in**>>* *(*p. *200)* Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

FooTypeSupport::create_data (p. 1512)

6.235.2.7 static DDS_ReturnCode_t FooTypeSupport::delete_data_ex (Foo * *a_data*, DDS_Boolean *deletePointers*) [static]

*<<**eXtension**>>* *(*p. *199)* Destroy a user data type instance.

The *generated* implementation of the operation knows how to destroy a data type and return all resources.

When `deletePointers` is **DDS_BOOLEAN_TRUE** (p. 298), all the references (pointers) are destroyed as well.

Parameters:

a_data *<<**in**>>* *(*p. *200)* Cannot be NULL.

deletePointers *<<**in**>>* *(*p. *200)* Whether or not to destroy pointers.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

FooTypeSupport::create_data_ex (p. 1513)

6.235.2.8 static DDS_ReturnCode_t FooTypeSupport::initialize_data (Foo * *a_data*) [static]

*<<**eXtension**>>* *(p. 199)* Initialize data type.

The *generated* implementation of the operation knows how to initialize a data type. This method is typically called to initialize a data type that is allocated on the stack.

Parameters:

a_data *<<**inout**>>* *(p. 200)* Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

FooTypeSupport::finalize_data (p. 1516)

6.235.2.9 static DDS_ReturnCode_t FooTypeSupport::initialize_data_ex (Foo * *a_data*, DDS_Boolean *allocatePointers*) [static]

*<<**eXtension**>>* *(p. 199)* Initialize data type.

The *generated* implementation of the operation knows how to initialize a data type. This method is typically called to initialize a data type that is allocated on the stack.

When `allocatePointers` is **DDS_BOOLEAN_TRUE** (p. 298), all the references (pointers) in the type are recursively allocated.

Parameters:

a_data *<<**inout**>>* *(p. 200)* Cannot be NULL.

allocatePointers *<<**in**>>* *(p. 200)* Whether or not to recursively allocate pointers.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

FooTypeSupport::finalize_data_ex (p. 1516)

6.235.2.10 static DDS_ReturnCode_t FooTypeSupport::finalize_data (Foo * *a_data*) [static]

<<*eXtension*>> *(*p. *199)* Finalize data type.

The *generated* implementation of the operation knows how to finalize a data type. This method is typically called to finalize a data type that has previouslly been initialized.

Parameters:

a_data <<*in*>> *(*p. *200)* Cannot be NULL.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

FooTypeSupport::initialize_data (p. 1515)

6.235.2.11 static DDS_ReturnCode_t FooTypeSupport::finalize_data_ex (Foo * *a_data*, DDS_Boolean *deletePointers*) [static]

<<*eXtension*>> *(*p. *199)* Finalize data type.

The *generated* implementation of the operation knows how to finalize a data type. This method is typically called to finalize a data type that has previouslly been initialized.

When `deletePointers` is **DDS_BOOLEAN_TRUE** (p. 298), the memory required by the references (pointers) associated to the type is freed.

Parameters:

a_data <<*in*>> *(*p. *200)* Cannot be NULL.

deletePointers <<*in*>> *(*p. *200)* Whether or not to free memory allocated by the pointers.

Returns:

One of the **Standard Return Codes** (p. 314)

See also:

FooTypeSupport::initialize_data_ex (p. 1515)

6.235.2.12 static const char* FooTypeSupport::get_type_name () [static]

Get the default name for this type.

Can be used for calling **FooTypeSupport::register_type** (p. 1510) or creating **DDSTopic** (p. 1419)

Returns:

default name for this type

See also:

FooTypeSupport::register_type (p. 1510)
DDSDomainParticipant::create_topic (p. 1175)

6.235.2.13 static void FooTypeSupport::print_data (const Foo * *a_data*) [static]

<<***eXtension***>> *(p. 199)* Print value of data type to standard out.

The *generated* implementation of the operation knows how to print value of a data type.

Parameters:

a_data <<***in***>> *(p. 200)* Data type to be printed.

6.236 NDDS_Config_LibraryVersion_t Struct Reference

The version of a single library shipped as part of an RTI Connext distribution.

Public Attributes

- **DDS_Long major**
  The major version of a single RTI Connext library.
- **DDS_Long minor**
  The minor version of a single RTI Connext library.
- char **release**
  The release letter of a single RTI Connext library.
- **DDS_Long build**
  The build number of a single RTI Connext library.

6.236.1 Detailed Description

The version of a single library shipped as part of an RTI Connext distribution.

RTI Connext is comprised of a number of separate libraries. Although RTI Connext as a whole has a version, the individual libraries each have their own versions as well. It may be necessary to check these individual library versions when seeking technical support.

6.236.2 Member Data Documentation

6.236.2.1 DDS_Long NDDS_Config_LibraryVersion_t::major

The major version of a single RTI Connext library.

6.236.2.2 DDS_Long NDDS_Config_LibraryVersion_t::minor

The minor version of a single RTI Connext library.

6.236.2.3 char NDDS_Config_LibraryVersion_t::release

The release letter of a single RTI Connext library.

6.236.2.4 DDS_Long NDDS_Config_LibraryVersion_t::build

The build number of a single RTI Connext library.

6.237 NDDS_Config_LogMessage Struct Reference

Log message.

Public Attributes

- const char * **text**
  Message text.
- **NDDS_Config_LogLevel level**

6.237.1 Detailed Description

Log message.

6.237.2 Member Data Documentation

6.237.2.1 const char* NDDS_Config_LogMessage::text

Message text.

6.237.2.2 NDDS_Config_LogLevel NDDS_Config_LogMessage::level

6.238 NDDS_Transport_Address_t Struct Reference

Addresses are stored individually as network-ordered bytes.

Public Attributes

- unsigned char **network_ordered_value** [NDDS_TRANSPORT_ADDRESS_LENGTH]

6.238.1 Detailed Description

Addresses are stored individually as network-ordered bytes.

RTI Connext addresses are numerically stored in a transport independent manner. RTI Connext uses a IPv6-compatible format, which means that the data structure to hold an **NDDS_Transport_Address_t** (p. 1521) is the same size as a data structure needed to hold an IPv6 address.

In addition, the functions provided to translate a string representation of an RTI Connext address to a value assumes that the string presentation follows the IPv6 address presentation as specified in RFC 2373.

An **NDDS_Transport_Address_t** (p. 1521) always stores the address in network-byte order (which is Big Endian).

For example, IPv4 multicast address of 225.0.0.0 is represented by

{{0,0,0,0, 0,0,0,0, 0,0,0,0, 0xE1,0,0,0}} regardless of endianness,

where 0xE1 is the 13th byte of the structure (`network_ordered_value`[12]).

6.238.2 Member Data Documentation

6.238.2.1 unsigned char NDDS_Transport_Address_t::network_ordered_value[NDDS_TRANSPORT_ADDRESS_LENGTH]

network-byte ordered (i.e., bit 0 is the most significant bit and bit 128 is the least significant bit).

6.239 NDDS_Transport_Property_t Struct Reference

Base structure that must be inherited by derived Transport Plugin classes.

Public Attributes

- **NDDS_Transport_ClassId_t classid**

  The Transport-Plugin Class ID.

- RTI_INT32 **address_bit_count**

  Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128.

- RTI_INT32 **properties_bitmap**

  A bitmap that defines various properties of the transport to the RTI Connext core.

- RTI_INT32 **gather_send_buffer_count_max**

  Specifies the maximum number of buffers that RTI Connext can pass to the `send()` *method of a transport plugin.*

- RTI_INT32 **message_size_max**

  The maximum size of a message in bytes that can be sent or received by the transport plugin.

- char ** **allow_interfaces_list**

  A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `allow_interfaces_list_length` *> 0), allow the use of only these interfaces. If the list is empty, allow the use of all interfaces.*

- RTI_INT32 **allow_interfaces_list_length**

  Number of elements in the `allow_interfaces_list`*.*

- char ** **deny_interfaces_list**

  A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `deny_interfaces_list_length` *> 0), deny the use of these interfaces.*

- RTI_INT32 **deny_interfaces_list_length**

  Number of elements in the `deny_interfaces_list`*.*

^ char ** **allow_multicast_interfaces_list**

A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `allow_multicast_interfaces_list_length` *> 0), allow the use of multicast only on these interfaces; otherwise allow the use of all the allowed interfaces.*

^ RTI_INT32 **allow_multicast_interfaces_list_length**

Number of elements in the `allow_multicast_interfaces_list`.

^ char ** **deny_multicast_interfaces_list**

A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `deny_multicast_interfaces_list_length` *> 0), deny the use of those interfaces for multicast.*

^ RTI_INT32 **deny_multicast_interfaces_list_length**

Number of elements in `deny_multicast_interfaces_list`.

6.239.1 Detailed Description

Base structure that must be inherited by derived Transport Plugin classes.

This structure contains properties that must be set before registration of any transport plugin with RTI Connext. The RTI Connext core will configure itself to use the plugin based on the properties set within this structure.

A transport plugin may extend from this structure to add transport-specific properties.

In the C-language, this can be done by creating a custom plugin property structure whose first member is a **NDDS_Transport_Property_t** (p. 1522) structure.

For example,

```
struct MyTransport_Plugin_Property_t {

    NDDS_Transport_Property_t base_properties;

    int myIntProperty;
    < etc >;
};
```

WARNING: The transport properties of an instance of a Transport Plugin should be considered immutable after the plugin has been created. That means the values contained in the property structure stored as a part of the transport plugin itself should not be changed. If those values are modified, the results are undefined.

6.239.2 Member Data Documentation

6.239.2.1 NDDS_Transport_ClassId_t NDDS_Transport_Property_t::classid

The Transport-Plugin Class ID.

Assigned by the implementor of the transport plugin, Class ID's below **NDDS_TRANSPORT_CLASSID_RESERVED_RANGE** (p. 249) are reserved for RTI (Real-Time Innovations) usage.

User-defined transports should set an ID above this range.

The ID should be globally unique for each Transport-Plugin class. Transport-Plugin implementors should ensure that the class IDs do not conflict with each other amongst different Transport-Plugin classes.

Invariant:

The `classid` is invariant for the lifecycle of a transport plugin.

6.239.2.2 RTI_INT32 NDDS_Transport_Property_t::address_bit_count

Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128.

A transport plugin should define the range of addresses (starting from 0x0) that are meaningful to the plugin. It does this by setting the number of bits of an IPv6 address that will be used to designate an address in the network to which the transport plugin is connected.

For example, for an address range of 0-255, the `address_bit_count` should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.

See also:

Transport Class Attributes (p. 126)

6.239.2.3 RTI_INT32 NDDS_Transport_Property_t::properties_bitmap

A bitmap that defines various properties of the transport to the RTI Connext core.

Currently, the only property supported is whether or not the transport plugin will always loan a buffer when RTI Connext tries to receive a message using the plugin. This is in support of a zero-copy interface.

See also:

NDDS_TRANSPORT_PROPERTY_BIT_BUFFER_ALWAYS_LOANED (p. 255)

6.239.2.4 RTI_INT32 NDDS_Transport_Property_t::gather_send_buffer_count_max

Specifies the maximum number of buffers that RTI Connext can pass to the `send()` method of a transport plugin.

The transport plugin `send()` API supports a gather-send concept, where the `send()` call can take several discontiguous buffers, assemble and send them in a single message. This enables RTI Connext to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.

However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent RTI Connext from trying to gather too many buffers into a send call for the transport plugin.

RTI Connext requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is defined to be **NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN** (p. 256).

If the underlying transport does not support a gather-send concept directly, then the transport plugin itself must copy the separate buffers passed into the `send()` call into a single buffer for sending or otherwise send each buffer individually. However this is done by the transport plugin, the `receive_rEA()` call of the destination application should assemble, if needed, all of the pieces of the message into a single buffer before the message is passed to the RTI Connext layer.

6.239.2.5 RTI_INT32 NDDS_Transport_Property_t::message_size_max

The maximum size of a message in bytes that can be sent or received by the transport plugin.

If the maximum size of a message that can be sent by a transport plugin is user configurable, the transport plugin should provide a default value for this

property. In any case, this value must be set before the transport plugin is registered, so that RTI Connext can properly use the plugin.

Note:

- If this value is increased from the default for any of the built-in transports, or if custom transports are used, then the **DDS_ReceiverPoolQosPolicy::buffer_size** (p. 864) on the **DDSDomainParticipant** (p. 1139) should also be changed.

6.239.2.6 char** NDDS_Transport_Property_t::allow_interfaces_list

A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `allow_interfaces_list_length` > 0), allow the use of only these interfaces. If the list is empty, allow the use of all interfaces.

The "white" list restricts *reception* to a particular set of interfaces for unicast UDP.

Multicast output will be sent and may be received over the interfaces in the list.

It is up to the transport plugin to interpret the list of strings passed in.

For example, the following are acceptable strings in IPv4 format: 192.168.1.1, 192.168.1.*, 192.168.*, 192.*, ether0

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

The caller (user) must manage the memory of the list. The memory may be freed after the **DDSDomainParticipant** (p. 1139) is deleted.

6.239.2.7 RTI_INT32 NDDS_Transport_Property_t::allow_interfaces_list_length

Number of elements in the `allow_interfaces_list`.

By default, `allow_interfaces_list_length` = 0, i.e. an empty list.

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

6.239.2.8 char** NDDS_Transport_Property_t::deny_interfaces_list

A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `deny_interfaces_list_length` > 0), deny

the use of these interfaces.

This "black" list is applied *after* the `allow_interfaces_list` and filters out the interfaces that should not be used.

The resulting list restricts *reception* to a particular set of interfaces for unicast UDP. Multicast output will be sent and may be received over the interfaces in the list.

It is up to the transport plugin to interpret the list of strings passed in.

For example, the following are acceptable strings in IPv4 format: 192.168.1.1, 192.168.1.∗, 192.168.∗, 192.∗, ether0

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

The caller (user) must manage the memory of the list. The memory may be freed after the **DDSDomainParticipant** (p. 1139) is deleted.

6.239.2.9 RTI_INT32 NDDS_Transport_Property_t::deny_interfaces_list_length

Number of elements in the `deny_interfaces_list`.

By default, `deny_interfaces_list_length` = 0 (i.e., an empty list).

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

6.239.2.10 char∗∗ NDDS_Transport_Property_t::allow_multicast_interfaces_list

A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `allow_multicast_interfaces_list_length` > 0), allow the use of multicast only on these interfaces; otherwise allow the use of all the allowed interfaces.

This "white" list sub-selects from the allowed interfaces obtained *after* applying the `allow_interfaces_list` "white" list *and* the `deny_interfaces_list` "black" list.

After `allow_multicast_interfaces_list`, the `deny_multicast_interfaces_list` is applied. Multicast output will be sent and may be received over the interfaces in the resulting list.

If this list is empty, all the allowed interfaces will be potentially used for multicast. It is up to the transport plugin to interpret the list of strings passed

in.

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

The caller (user) must manage the memory of the list. The memory may be freed after the **DDSDomainParticipant** (p. 1139) is deleted.

6.239.2.11 RTI_INT32 NDDS_Transport_Property_t::allow_multicast_interfaces_list_length

Number of elements in the `allow_multicast_interfaces_list`.

By default, `allow_multicast_interfaces_list_length` = 0 (i.e., an empty list).

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

6.239.2.12 char** NDDS_Transport_Property_t::deny_multicast_interfaces_list

A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty (i.e., `deny_multicast_interfaces_list_length` > 0), deny the use of those interfaces for multicast.

This "black" list is applied after `allow_multicast_interfaces_list` and filters out interfaces that should not be used for multicast.

Multicast output will be sent and may be received over the interfaces in the resulting list.

It is up to the transport plugin to interpret the list of strings passed in.

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

The caller (user) must manage the memory of the list. The memory may be freed after the **DDSDomainParticipant** (p. 1139) is deleted.

6.239.2.13 RTI_INT32 NDDS_Transport_Property_t::deny_multicast_interfaces_list_length

Number of elements in `deny_multicast_interfaces_list`.

By default, `deny_multicast_interfaces_list_length` = 0 (i.e., an empty list).

This property is not interpreted by the RTI Connext core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.

6.240 NDDS_Transport_Shmem_Property_t Struct Reference

Subclass of **NDDS_Transport_Property_t** (p. 1522) allowing specification of parameters that are specific to the shared-memory transport.

Public Attributes

- struct **NDDS_Transport_Property_t parent**
  Generic properties of all transport plugins.
- RTI_INT32 **received_message_count_max**
  Number of messages that can be buffered in the receive queue.
- RTI_INT32 **receive_buffer_size**
  The total number of bytes that can be buffered in the receive queue.

6.240.1 Detailed Description

Subclass of **NDDS_Transport_Property_t** (p. 1522) allowing specification of parameters that are specific to the shared-memory transport.

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564)

6.240.2 Member Data Documentation

6.240.2.1 struct NDDS_Transport_Property_t NDDS_Transport_Shmem_Property_t::parent [read]

Generic properties of all transport plugins.

6.240.2.2 RTI_INT32 NDDS_Transport_Shmem_Property_t::received_message_count_max

Number of messages that can be buffered in the receive queue.

This does not guarantee that the Transport-Plugin will actually be able to buffer `received_message_count_max` messages of the maximum size set in **NDDS_Transport_Property_t::message_size_max** (p. 1525). The total number of

bytes that can be buffered for a transport plug-in is actually controlled by `receive_buffer_size`.

See also:

NDDS_Transport_Property_t (p. 1522), **NDDS_TRANSPORT_SHMEM_RECEIVED_MESSAGE_COUNT_MAX_DEFAULT** (p. 262)

6.240.2.3 RTI_INT32 NDDS_Transport_Shmem_Property_t::receive_buffer_size

The total number of bytes that can be buffered in the receive queue.

This number controls how much memory is allocated by the plugin for the receive queue. The actual number of bytes allocated is:

```
size = receive_buffer_size + message_size_max +
                    received_message_count_max * fixedOverhead
```

where `fixedOverhead` is some small number of bytes used by the queue data structure. The following rules are noted:

- `receive_buffer_size` < `message_size_max` * `received_message_count_max`, then the transport plugin will not be able to store `received_message_count_max` messages of size `message_size_max`.
- `receive_buffer_size` > `message_size_max` * `received_message_count_max`, then there will be memory allocated that cannot be used by the plugin and thus wasted.

To optimize memory usage, the user is allowed to specify a size for the receive queue to be less than that required to hold the maximum number of messages which are all of the maximum size.

In most situations, the average message size may be far less than the maximum message size. So for example, if the maximum message size is 64 K bytes, and the user configures the plugin to buffer at least 10 messages, then 640 K bytes of memory would be needed if all messages were 64 K bytes. Should this be desired, then `receive_buffer_size` should be set to 640 K bytes.

However, if the average message size is only 10 K bytes, then the user could set the `receive_buffer_size` to 100 K bytes. This allows the user to optimize the memory usage of the plugin for the average case and yet allow the plugin to handle the extreme case.

NOTE, the queue will always be able to hold 1 message of `message_size_max` bytes, no matter what the value of `receive_buffer_size` is.

See also:

NDDS_TRANSPORT_SHMEM_RECEIVE_BUFFER_SIZE_DEFAULT (p. 262)

6.241 NDDS_Transport_UDPv4_Property_t Struct Reference

Configurable IPv4/UDP Transport-Plugin properties.

Public Attributes

- struct **NDDS_Transport_Property_t parent**

  Generic properties of all transport plugins.

- RTI_INT32 **send_socket_buffer_size**

  Size in bytes of the send buffer of a socket used for sending.

- RTI_INT32 **recv_socket_buffer_size**

  Size in bytes of the receive buffer of a socket used for receiving.

- RTI_INT32 **unicast_enabled**

  Allows the transport plugin to use unicast for sending and receiving.

- RTI_INT32 **multicast_enabled**

  Allows the transport plugin to use multicast for sending and receiving.

- RTI_INT32 **multicast_ttl**

  Value for the time-to-live parameter for all multicast sends using this plugin.

- RTI_INT32 **multicast_loopback_disabled**

  Prevents the transport plugin from putting multicast packets onto the loopback interface.

- RTI_INT32 **ignore_loopback_interface**

  Prevents the transport plugin from using the IP loopback interface.

- RTI_INT32 **ignore_nonup_interfaces**

  Prevents the transport plugin from using a network interface that is not reported as UP by the operating system.

- RTI_INT32 **ignore_nonrunning_interfaces**

  Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.

- RTI_INT32 **no_zero_copy**

  Prevents the transport plugin from doing a zero copy.

- RTI_INT32 **send_blocking**

  Control blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.

- RTI_UINT32 **transport_priority_mask**

  Set mask for use of transport priority field.

- RTI_INT32 **transport_priority_mapping_low**

  Set low value of output range to IPv4 TOS.

- RTI_INT32 **transport_priority_mapping_high**

  Set high value of output range to IPv4 TOS.

- RTI_UINT32 **interface_poll_period**

  Specifies the period in milliseconds to query for changes in the state of all the interfaces.

- RTI_INT32 **reuse_multicast_receive_resource**

  Controls whether or not to reuse multicast receive resources.

- RTI_INT32 **protocol_overhead_max**

  Maximum size in bytes of protocol overhead, including headers.

6.241.1 Detailed Description

Configurable IPv4/UDP Transport-Plugin properties.

The properties in this structure can be modified by the end user to configure the plugin. However, the properties must be set before the plugin is instantiated.

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564)
NDDS_Transport_UDPv4_new (p. 270)

6.241.2 Member Data Documentation

6.241.2.1 struct NDDS_Transport_Property_t NDDS_Transport_UDPv4_Property_t::parent [read]

Generic properties of all transport plugins.

6.241.2.2 RTI_INT32 NDDS_Transport_UDPv4_Property_t::send_socket_buffer_size

Size in bytes of the send buffer of a socket used for sending.

On most operating systems, `setsockopt()` will be called to set the SENDBUF to the value of this parameter.

This value must be greater than or equal to **NDDS_Transport_Property_t::message_size_max** (p. 1525). The maximum value is operating system-dependent.

By default, it will be set to be **NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT** (p. 269).

If users configure this parameter to be **NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT** (p. 269), then `setsockopt()` (or equivalent) will not be called to size the send buffer of the socket.

See also:

NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT (p. 269)
NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT (p. 269)

6.241.2.3 RTI_INT32 NDDS_Transport_UDPv4_Property_t::recv_socket_buffer_size

Size in bytes of the receive buffer of a socket used for receiving.

On most operating systems, `setsockopt()` will be called to set the RECVBUF to the value of this parameter.

This value must be greater than or equal to **NDDS_Transport_Property_t::message_size_max** (p. 1525). The maximum value is operating system-dependent.

By default, it will be set to be **NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT** (p. 269).

If it is set to **NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT** (p. 269), then `setsockopt()` (or equivalent) will not be called to size the receive buffer of the socket.

See also:

NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT (p. 269)

NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_-DEFAULT (p. 269)

6.241.2.4 RTI_INT32 NDDS_Transport_UDPv4_Property_-t::unicast_enabled

Allows the transport plugin to use unicast for sending and receiving.

This value turns unicast UDP on (if set to 1) or off (if set to 0) for this plugin. By default, it will be turned on (1). Also by default, the plugin will use all the allowed network interfaces that it finds up and running when the plugin is instanced.

6.241.2.5 RTI_INT32 NDDS_Transport_UDPv4_Property_-t::multicast_enabled

Allows the transport plugin to use multicast for sending and receiving.

This value turns multicast UDP on (if set to 1) or off (if set to 0) for this plugin. By default, it will be turned on (1) for those platforms that support multicast. Also by default, the plugin will use the all network interfaces allowed for multicast that it finds up and running when the plugin is instanced.

6.241.2.6 RTI_INT32 NDDS_Transport_UDPv4_Property_-t::multicast_ttl

Value for the time-to-live parameter for all multicast sends using this plugin.

This value is used to set the TTL of multicast packets sent by this transport plugin.

See also:

NDDS_TRANSPORT_UDPV4_MULTICAST_TTL_DEFAULT (p. 269)

6.241.2.7 RTI_INT32 NDDS_Transport_UDPv4_Property_-t::multicast_loopback_disabled

Prevents the transport plugin from putting multicast packets onto the loopback interface.

If multicast loopback is disabled (this value is set to 1), then when sending multicast packets, RTI Connext will *not* put a copy of the packets on the loopback

interface. This prevents applications on the same node (including itself) from receiving those packets.

This value is set to 0 by default, meaning multicast loopback is *enabled*.

Disabling multicast loopback (setting this value to 1) may result in minor performance gains when using multicast.

[NOTE: Windows CE systems do not support multicast loopback. This field is ignored for Windows CE targets.]

6.241.2.8 RTI_INT32 NDDS_Transport_UDPv4_Property_t::ignore_loopback_interface

Prevents the transport plugin from using the IP loopback interface.

Currently three values are allowed:

- **0**: Forces local traffic to be sent over loopback, even if a more efficient transport (such as shared memory) is installed (in which case traffic will be sent over both transports).
- **1**: Disables local traffic via this plugin. The IP loopback interface is not used, even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plugin (such as shared memory) instead of the IP loopback.
- **-1**: Automatic. Lets RTI Connext decide between the above two choices.

The current "automatic" (-1) RTI Connext policy is as follows.

- If a shared memory transport plugin is available for local traffic, the effective value is 1 (i.e., disable UPV4 local traffic).
- Otherwise, the effective value is 0 (i.e., use UDPv4 for local traffic also).

[**default**] -1 Automatic RTI Connext policy based on availability of the shared memory transport.

6.241.2.9 RTI_INT32 NDDS_Transport_UDPv4_Property_t::ignore_nonup_interfaces

Prevents the transport plugin from using a network interface that is not reported as UP by the operating system.

The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not

be used. This property allows the user to configure the transport to start using even the interfaces which were not reported as UP.

Two values are allowed:

- **0**: Allow the use of interfaces which were not reported as UP.
- **1**: Do not use interfaces which were not reported as UP.

[**default**] 1

6.241.2.10 RTI_INT32 NDDS_Transport_UDPv4_Property_t::ignore_nonrunning_interfaces

Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.

The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged.

Two values are allowed:

- **0**: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.
- **1**: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.

[**default**] 0 (i.e., do not check RUNNING flag)

6.241.2.11 RTI_INT32 NDDS_Transport_UDPv4_Property_t::no_zero_copy

Prevents the transport plugin from doing a zero copy.

By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application.

The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the H/W, device driver, or the OS to allow the zero copy feature to work for your application, you may have no choice but to turn off zero copy use.

By default this is set to 0, so RTI Connext will use the zero-copy API if offered by the OS.

6.241.2.12 RTI_INT32 NDDS_Transport_UDPv4_Property_t::send_blocking

Control blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.

Currently two values are defined:

- **NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS:** Sockets are blocking (default socket options for Operating System).
- **NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER:** Sockets are modified to make them non-blocking. THIS IS NOT A SUPPORTED CONFIGURATION AND MAY CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.

[**default**] NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS.

6.241.2.13 RTI_UINT32 NDDS_Transport_UDPv4_Property_t::transport_priority_mask

Set mask for use of transport priority field.

This is used in conjunction with **NDDS_Transport_UDPv4_Property_t::transport_priority_mapping_low** (p. 1540) and **NDDS_Transport_UDPv4_Property_t::transport_priority_mapping_high** (p. 1540) to define the mapping from DDS transport priority (see **TRANSPORT_PRIORITY** (p. 373)) to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket.

For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high.

If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.

[**default**] 0.

6.241.2.14 RTI_INT32 NDDS_Transport_UDPv4_Property_t::transport_priority_mapping_low

Set low value of output range to IPv4 TOS.

This is used in conjunction with **NDDS_Transport_UDPv4_Property_t::transport_priority_mask** (p. 1539) and **NDDS_Transport_UDPv4_Property_t::transport_priority_mapping_high** (p. 1540) to define the mapping from DDS transport priority to the IPv4 TOS field. Defines the low value of the output range for scaling.

Note that IPv4 TOS is generally an 8-bit value.

[default] 0.

6.241.2.15 RTI_INT32 NDDS_Transport_UDPv4_Property_t::transport_priority_mapping_high

Set high value of output range to IPv4 TOS.

This is used in conjunction with **NDDS_Transport_UDPv4_Property_t::transport_priority_mask** (p. 1539) and **NDDS_Transport_UDPv4_Property_t::transport_priority_mapping_low** (p. 1540) to define the mapping from DDS transport priority to the IPv4 TOS field. Defines the high value of the output range for scaling.

Note that IPv4 TOS is generally an 8-bit value.

[default] 0xff.

6.241.2.16 RTI_UINT32 NDDS_Transport_UDPv4_Property_t::interface_poll_period

Specifies the period in milliseconds to query for changes in the state of all the interfaces.

The value of this property is ignored if ignore_non_interfaces is 1. If ignore_nonup_interfaces is 0 then the UDPv4 transport creates a new thread to query the status of the interfaces. This property specifies the polling period in milliseconds for performing this query.

[default] 500 milliseconds.

6.241.2.17 RTI_INT32 NDDS_Transport_UDPv4_Property_t::reuse_multicast_receive_resource

Controls whether or not to reuse multicast receive resources.

Setting this to 0 (FALSE) prevents multicast crosstalk by uniquely configuring a port and creating a receive thread for each multicast group address.

[**default**] 0.

6.241.2.18 RTI_INT32 NDDS_Transport_UDPv4_Property_t::protocol_overhead_max

Maximum size in bytes of protocol overhead, including headers.

This value is the maximum size, in bytes, of protocol-related overhead. Normally, the overhead accounts for UDP and IP headers. The default value is set to accommodate the most common UDP/IP header size.

Note that when **NDDS_Transport_Property_t::message_size_max** (p. 1525) plus this overhead is larger than the UDPv4 maximum message size (65535 bytes), the middleware will automatically reduce the effective message_size_max, to 65535 minus this overhead.

[**default**] 28.

See also:

NDDS_Transport_Property_t::message_size_max (p. 1525)

6.242 NDDS_Transport_UDPv6_Property_t Struct Reference

Configurable IPv6/UDP Transport-Plugin properties.

Public Attributes

- struct **NDDS_Transport_Property_t parent**

  Generic properties of all transport plugins.

- RTI_INT32 **send_socket_buffer_size**

  Size in bytes of the send buffer of a socket used for sending.

- RTI_INT32 **recv_socket_buffer_size**

  Size in bytes of the receive buffer of a socket used for receiving.

- RTI_INT32 **unicast_enabled**

  Allows the transport plugin to use unicast for sending and receiving.

- RTI_INT32 **multicast_enabled**

  Allows the transport plugin to use multicast for sending and receiving.

- RTI_INT32 **multicast_ttl**

  Value for the time-to-live parameter for all multicast sends using this plugin.

- RTI_INT32 **multicast_loopback_disabled**

  Prevents the transport plugin from putting multicast packets onto the loopback interface.

- RTI_INT32 **ignore_loopback_interface**

  Prevents the transport plugin from using the IP loopback interface.

- RTI_INT32 **ignore_nonrunning_interfaces**

  Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.

- RTI_INT32 **no_zero_copy**

  Prevents the transport plugin from doing zero copy.

- RTI_INT32 **send_blocking**

  Control blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.

- RTI_INT32 **enable_v4mapped**

  Specify whether UDPv6 transport will process IPv4 addresses.

- RTI_UINT32 **transport_priority_mask**

  Set mask for use of transport priority field.

- RTI_INT32 **transport_priority_mapping_low**

  Set low value of output range to IPv6 TCLASS.

- RTI_INT32 **transport_priority_mapping_high**

  Set high value of output range to IPv6 TCLASS.

6.242.1 Detailed Description

Configurable IPv6/UDP Transport-Plugin properties.

The properties in this structure can be modified by the end user to configure the plugin. However, the properties must be set before the plugin is instantiated.

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564)
NDDS_Transport_UDPv6_new (p. 280)

6.242.2 Member Data Documentation

6.242.2.1 struct NDDS_Transport_Property_t NDDS_Transport_UDPv6_Property_t::parent [read]

Generic properties of all transport plugins.

6.242.2.2 RTI_INT32 NDDS_Transport_UDPv6_Property_t::send_socket_buffer_size

Size in bytes of the send buffer of a socket used for sending.

On most operating systems, `setsockopt()` will be called to set the SENDBUF to the value of this parameter.

This value must be greater than or equal to **NDDS_Transport_Property_t::message_size_max** (p. 1525). The maximum value is operating system-dependent.

By default, it will be set to be **NDDS_TRANSPORT_UDPV6_-MESSAGE_SIZE_MAX_DEFAULT** (p. 279).

If users configure this parameter to be **NDDS_TRANSPORT_UDPV6_-SOCKET_BUFFER_SIZE_OS_DEFAULT** (p. 279), then `setsockopt()` (or equivalent) will not be called to size the send buffer of the socket.

See also:

NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_-DEFAULT (p. 279)
NDDS_TRANSPORT_UDPV6_SOCKET_BUFFER_SIZE_OS_-DEFAULT (p. 279)

6.242.2.3 RTI_INT32 NDDS_Transport_UDPv6_Property_t::recv_-socket_buffer_size

Size in bytes of the receive buffer of a socket used for receiving.

On most operating systems, `setsockopt()` will be called to set the RECVBUF to the value of this parameter.

This value must be greater than or equal to **NDDS_Transport_Property_-t::message_size_max** (p. 1525). The maximum value is operating system-dependent.

By default, it will be set to be **NDDS_TRANSPORT_UDPV6_-MESSAGE_SIZE_MAX_DEFAULT** (p. 279).

If it is set to **NDDS_TRANSPORT_UDPV6_SOCKET_BUFFER_-SIZE_OS_DEFAULT** (p. 279), then `setsockopt()` (or equivalent) will not be called to size the receive buffer of the socket.

See also:

NDDS_TRANSPORT_UDPV6_MESSAGE_SIZE_MAX_-DEFAULT (p. 279)
NDDS_TRANSPORT_UDPV6_SOCKET_BUFFER_SIZE_OS_-DEFAULT (p. 279)

6.242.2.4 RTI_INT32 NDDS_Transport_UDPv6_Property_-t::unicast_enabled

Allows the transport plugin to use unicast for sending and receiving.

This value turns unicast UDP on (if set to 1) or off (if set to 0) for this plugin. By default, it will be turned on (1). Also by default, the plugin will use all

the allowed network interfaces that it finds up and running when the plugin is instanced.

6.242.2.5 RTI_INT32 NDDS_Transport_UDPv6_Property_t::multicast_enabled

Allows the transport plugin to use multicast for sending and receiving.

This value turns multicast UDP on (if set to 1) or off (if set to 0) for this plugin. By default, it will be turned on (1) for those platforms that support multicast. Also by default, the plugin will use the all network interfaces allowed for multicast that it finds up and running when the plugin is instanced.

6.242.2.6 RTI_INT32 NDDS_Transport_UDPv6_Property_t::multicast_ttl

Value for the time-to-live parameter for all multicast sends using this plugin.

This is used to set the TTL of multicast packets sent by this transport plugin.

See also:

NDDS_TRANSPORT_UDPV6_MULTICAST_TTL_DEFAULT (p. 279)

6.242.2.7 RTI_INT32 NDDS_Transport_UDPv6_Property_t::multicast_loopback_disabled

Prevents the transport plugin from putting multicast packets onto the loopback interface.

If multicast loopback is disabled (this value is set to 1), then when sending multicast packets, RTI Connext will *not* put a copy of the packets on the loopback interface. This prevents applications on the same node (including itself) from receiving those packets.

This value is set to 0 by default, meaning multicast loopback is *enabled*.

Disabling multicast loopback (setting this value to 1) may result in minor performance gains when using multicast.

6.242.2.8 RTI_INT32 NDDS_Transport_UDPv6_Property_t::ignore_loopback_interface

Prevents the transport plugin from using the IP loopback interface.

Currently three values are allowed:

- **0**: Forces local traffic to be sent over loopback, even if a more efficient transport (such as shared memory) is installed (in which case traffic will be sent over both transports).

- **1**: Disables local traffic via this plugin. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient transport (such as shared memory) instead of the IP loopback.

- **-1**: Automatic. Lets RTI Connext decide between the above two choices.

The current "automatic" (-1) RTI Connext policy is as follows.

- If a shared memory transport plugin is available for local traffic, the effective value is 1 (i.e., disable UDPv6 local traffic).
- Otherwise, the effective value is 0 (i.e., use UDPv6 for local traffic also).

[default] -1 Automatic RTI Connext policy based on availability of the shared memory transport.

6.242.2.9 RTI_INT32 NDDS_Transport_UDPv6_Property_t::ignore_nonrunning_interfaces

Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.

The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged.

Two values are allowed:

- **0**: Do not check the RUNNING flag when enumerating interfaces, just make sure interface is UP.

- **1**: Check flag when enumerating interfaces and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.

[default] 0 (i.e., do not check RUNNING flag)

6.242.2.10 RTI_INT32 NDDS_Transport_UDPv6_Property_t::no_zero_copy

Prevents the transport plugin from doing zero copy.

By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometimes tax the OS resources in a manner that cannot be overcome by the application.

The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. If you cannot reconfigure the H/W, device driver, or the OS to allow the zero copy feature to work for your application, you may have no choice but to turn off the use of zero copy.

By default this is set to 0, so RTI Connext will use the zero copy API if offered by the OS.

6.242.2.11 RTI_INT32 NDDS_Transport_UDPv6_Property_t::send_blocking

Control blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.

Currently two values are defined:

- **NDDS_TRANSPORT_UDPV6_BLOCKING_ALWAYS:** Sockets are blocking (default socket options for Operating System).
- **NDDS_TRANSPORT_UDPV6_BLOCKING_NEVER:** Sockets are modified to make them non-blocking. THIS IS NOT A SUPPORTED CONFIGURATION AND MAY CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.

[**default**] NDDS_TRANSPORT_UDPV6_BLOCKING_ALWAYS.

6.242.2.12 RTI_INT32 NDDS_Transport_UDPv6_Property_t::enable_v4mapped

Specify whether UDPv6 transport will process IPv4 addresses.

Set this to 1 to turn on processing of IPv4 addresses. Note that this may make it incompatible with use of the UDPv4 transport within the same domain participant.

[**default**] 0.

6.242.2.13 RTI_UINT32 NDDS_Transport_UDPv6_Property_t::transport_priority_mask

Set mask for use of transport priority field.

If transport priority mapping is supported on the platform, this mask is used in conjunction with **NDDS_Transport_UDPv6_Property_t::transport_priority_mapping_low** (p. 1548) and **NDDS_Transport_UDPv6_Property_t::transport_priority_mapping_high** (p. 1548) to define the mapping from DDS transport priority (see **TRANSPORT_PRIORITY** (p. 373)) to the IPv6 TCLASS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv6 TCLASS field on an outgoing socket. (See the `Getting Started Guide` to find out if the transport priority is supported on a specific platform.)

For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high.

If the mask is set to zero, then the transport will not set IPv6 TCLASS for send sockets.

[default] 0.

6.242.2.14 RTI_INT32 NDDS_Transport_UDPv6_Property_t::transport_priority_mapping_low

Set low value of output range to IPv6 TCLASS.

This is used in conjunction with **NDDS_Transport_UDPv6_Property_t::transport_priority_mask** (p. 1548) and **NDDS_Transport_UDPv6_Property_t::transport_priority_mapping_high** (p. 1548) to define the mapping from DDS transport priority to the IPv6 TCLASS field. Defines the low value of the output range for scaling.

Note that IPv6 TCLASS is generally an 8-bit value.

[default] 0.

6.242.2.15 RTI_INT32 NDDS_Transport_UDPv6_Property_t::transport_priority_mapping_high

Set high value of output range to IPv6 TCLASS.

This is used in conjunction with **NDDS_Transport_UDPv6_Property_t::transport_priority_mask** (p. 1548) and **NDDS_Transport_UDPv6_Property_t::transport_priority_mapping_low** (p. 1548) to define the mapping from DDS transport priority to the IPv6 TCLASS field. Defines the high value of the output range for scaling.

Note that IPv6 TCLASS is generally an 8-bit value.

[**default**] 0xff.

6.243 NDDSConfigLogger Class Reference

<<***interface***>> *(p. 199)* The singleton type used to configure RTI Connext logging.

Public Member Functions

- **NDDS_Config_LogVerbosity get_verbosity** ()

  Get the verbosity at which RTI Connext is currently logging diagnostic information.

- **NDDS_Config_LogVerbosity get_verbosity_by_category** (**NDDS_Config_LogCategory** category)

  Get the verbosity at which RTI Connext is currently logging diagnostic information in the given category.

- void **set_verbosity** (**NDDS_Config_LogVerbosity** verbosity)

  Set the verbosity at which RTI Connext will log diagnostic information.

- void **set_verbosity_by_category** (**NDDS_Config_LogCategory** category, **NDDS_Config_LogVerbosity** verbosity)

  Set the verbosity at which RTI Connext will log diagnostic information in the given category.

- FILE * **get_output_file** ()

  Get the file to which the logged output is redirected.

- bool **set_output_file** (FILE *out)

  Set the file to which the logged output is redirected.

- **NDDSConfigLoggerDevice** * **get_output_device** ()

  Return the user device registered with the logger.

- bool **set_output_device** (**NDDSConfigLoggerDevice** *device)

  *Register a **NDDSConfigLoggerDevice** (p. 1555).*

- **NDDS_Config_LogPrintFormat get_print_format** ()

  Get the current message format that RTI Connext is using to log diagnostic information.

- bool **set_print_format** (**NDDS_Config_LogPrintFormat** print_format)

Set the message format that RTI Connext will use to log diagnostic information.

Static Public Member Functions

- static **NDDSConfigLogger** * **get_instance** ()

  Get the singleton instance of this type.

- static void **finalize_instance** ()

  Finalize the singleton instance of this type.

6.243.1 Detailed Description

<<***interface***>> *(p. 199)* The singleton type used to configure RTI Connext logging.

6.243.2 Member Function Documentation

6.243.2.1 static NDDSConfigLogger* NDDSConfigLogger::get_instance () [static]

Get the singleton instance of this type.

6.243.2.2 static void NDDSConfigLogger::finalize_instance () [static]

Finalize the singleton instance of this type.

6.243.2.3 NDDS_Config_LogVerbosity NDDSConfigLogger::get_verbosity ()

Get the verbosity at which RTI Connext is currently logging diagnostic information.

The default verbosity if DDSLogger::set_verbosity is never called is DDS_NDDS_CONFIG_LOG_VERBOSITY_ERROR.

If DDSLogger::set_verbosity_by_category has been used to set different verbosities for different categories of messages, this method will return the maximum verbosity of all categories.

6.243.2.4 NDDS_Config_LogVerbosity NDDSConfigLogger::get_verbosity_by_category (NDDS_Config_LogCategory *category*)

Get the verbosity at which RTI Connext is currently logging diagnostic information in the given category.

The default verbosity if DDSLogger::set_verbosity and DDSLogger::set_verbosity_by_category are never called is DDS_NDDS_CONFIG_LOG_VERBOSITY_ERROR.

6.243.2.5 void NDDSConfigLogger::set_verbosity (NDDS_Config_LogVerbosity *verbosity*)

Set the verbosity at which RTI Connext will log diagnostic information.

Note: Logging at high verbosities will be detrimental to your application's performance. Your default setting should typically remain at DDS_NDDS_CONFIG_LOG_VERBOSITY_WARNING or below. (The default verbosity if you never set it is DDS_NDDS_CONFIG_LOG_VERBOSITY_ERROR.)

6.243.2.6 void NDDSConfigLogger::set_verbosity_by_category (NDDS_Config_LogCategory *category*, NDDS_Config_LogVerbosity *verbosity*)

Set the verbosity at which RTI Connext will log diagnostic information in the given category.

6.243.2.7 FILE* NDDSConfigLogger::get_output_file ()

Get the file to which the logged output is redirected.

If no output file has been registered through DDSLogger::set_output_file, this method will return NULL. In this case, logged output will on most platforms go to standard out as if through printf.

6.243.2.8 bool NDDSConfigLogger::set_output_file (FILE * *out*)

Set the file to which the logged output is redirected.

The file passed may be NULL, in which case further logged output will be redirected to the platform-specific default output location (standard out on most platforms).

6.243.2.9 NDDSConfigLoggerDevice* NDDSConfigLogger::get_output_device ()

Return the user device registered with the logger.

Returns:

Registered user device or NULL if no user device is registered.

6.243.2.10 bool NDDSConfigLogger::set_output_device (NDDSConfigLoggerDevice * *device*)

Register a **NDDSConfigLoggerDevice** (p. 1555).

Register the specified logging device with the logger.

There can be at most only one device registered with the logger at any given time.

When a device is installed, the logger will stop sending the log messages to the standard output and to the file set with DDSLogger::set_output_file.

To remove an existing device, use this method with NULL as the device parameter. After a device is removed the logger will continue sending log messages to the standard output and to the output file.

To replace an existing device with a new device, use this method providing the new device as the device parameter.

When a device is unregistered (by setting it to NULL), **NDDSConfigLoggerDevice** (p. 1555) calls the method **NDDSConfigLoggerDevice::close** (p. 1556).

Parameters:

device *<<**in**>>* *(p. 200)* Logging device.

6.243.2.11 NDDS_Config_LogPrintFormat NDDSConfigLogger::get_print_format ()

Get the current message format that RTI Connext is using to log diagnostic information.

If DDSLogger::set_print_format is never called, the default format is DDS_NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT.

6.243.2.12 bool NDDSConfigLogger::set_print_format (NDDS_Config_LogPrintFormat *print_format*)

Set the message format that RTI Connext will use to log diagnostic information.

6.244 NDDSConfigLoggerDevice Class Reference

<<*interface*>> *(*p. *199)* Logging device interface. Use for user-defined logging devices.

Public Member Functions

- virtual void **write** (const **NDDS_Config_LogMessage** *message)=0

  Write a log message to a specified logging device.

- virtual void **close** ()

  Close the logging device.

6.244.1 Detailed Description

<<*interface*>> *(*p. *199)* Logging device interface. Use for user-defined logging devices.

Interface for handling log messages.

By default, the logger sends the log messages generated by RTI Connext to the standard output.

You can use the method DDSLogger::set_output_file to redirect the log messages to a file.

To further customize the management of generated log messages, the logger offers the method DDSLogger::set_output_device that allows you to install a user-defined logging device.

The logging device installed by the user must implement this interface.

6.244.2 Member Function Documentation

6.244.2.1 virtual void NDDSConfigLoggerDevice::write (const NDDS_Config_LogMessage * *message*) [pure virtual]

Write a log message to a specified logging device.

Parameters:

message <<*in*>> *(*p. *200)* Message to log.

6.244.2.2 virtual void NDDSConfigLoggerDevice::close () [virtual]

Close the logging device.

6.245 NDDSConfigVersion Class Reference

<<*interface*>> *(p. 199)* The version of an RTI Connext distribution.

Public Member Functions

- const **DDS_ProductVersion_t** & **get_product_version** () const
  Get the RTI Connext product version.
- const **NDDS_Config_LibraryVersion_t** & **get_cpp_api_version** () const
  Get the version of the C++ API library.
- const **NDDS_Config_LibraryVersion_t** & **get_c_api_version** () const
  Get the version of the C API library.
- const **NDDS_Config_LibraryVersion_t** & **get_core_version** () const
  Get the version of the core library.
- const char * **to_string** () const
  Get this version in string form.

Static Public Member Functions

- static const **NDDSConfigVersion** & **get_instance** ()
  Get the singleton instance of this type.

6.245.1 Detailed Description

<<*interface*>> *(p. 199)* The version of an RTI Connext distribution.

The complete version is made up of the versions of the individual libraries that make up the product distribution.

6.245.2 Member Function Documentation

6.245.2.1 static const NDDSConfigVersion& NDDSConfigVersion::get_instance () [static]

Get the singleton instance of this type.

6.245.2.2 const DDS_ProductVersion_t& NDDSConfigVersion::get_product_version () const

Get the RTI Connext product version.

6.245.2.3 const NDDS_Config_LibraryVersion_t& NDDSConfigVersion::get_cpp_api_version () const

Get the version of the C++ API library.

6.245.2.4 const NDDS_Config_LibraryVersion_t& NDDSConfigVersion::get_c_api_version () const

Get the version of the C API library.

6.245.2.5 const NDDS_Config_LibraryVersion_t& NDDSConfigVersion::get_core_version () const

Get the version of the core library.

6.245.2.6 const char* NDDSConfigVersion::to_string () const

Get this version in string form.

Combine all of the constituent library versions into a single string.

The memory in which the string is stored in internal to this **NDDSConfigVersion** (p. 1557). The caller should not modify it.

6.246 NDDSTransportSupport Class Reference

<<*interface*>> *(*p. *199)* The utility class used to configure RTI Connext pluggable transports.

Static Public Member Functions

- static **NDDS_Transport_Handle_t register_transport** (**DDSDomainParticipant** *participant_in, NDDS_Transport_Plugin *transport_in, const **DDS_StringSeq** &aliases_in, const **NDDS_Transport_Address_t** &network_address_in)

  *Register a transport plugin for use with a **DDSDomainParticipant** (*p. *1139), assigning it a network_address.*

- static **NDDS_Transport_Handle_t lookup_transport** (**DDSDomainParticipant** *participant_in, **DDS_StringSeq** &aliases_out, **NDDS_Transport_Address_t** &network_address_out, NDDS_Transport_Plugin *transport_in)

  *Look up a transport plugin within a **DDSDomainParticipant** (*p. *1139).*

- static **DDS_ReturnCode_t add_send_route** (const **NDDS_Transport_Handle_t** &transport_handle_in, const **NDDS_Transport_Address_t** &address_range_in, **DDS_Long** address_range_bit_count_in)

  Add a route for outgoing messages.

- static **DDS_ReturnCode_t add_receive_route** (const **NDDS_Transport_Handle_t** &transport_handle_in, const **NDDS_Transport_Address_t** &address_range_in, **DDS_Long** address_range_bit_count_in)

  Add a route for incoming messages.

- static **DDS_ReturnCode_t get_builtin_transport_property** (**DDSDomainParticipant** *participant_in, **DDS_TransportBuiltinKind** builtin_transport_kind_in, struct **NDDS_Transport_Property_t** &builtin_transport_property_inout)

  Get the properties used to create a builtin transport plugin.

- static **DDS_ReturnCode_t set_builtin_transport_property** (**DDSDomainParticipant** *participant_in, **DDS_TransportBuiltinKind** builtin_transport_kind_in, const struct **NDDS_Transport_Property_t** &builtin_transport_property_in)

  Set the properties used to create a builtin transport plugin.

- static NDDS_Transport_Plugin * **get_transport_plugin** (**DDSDomainParticipant** *participant_in, const char *alias_in)

*Retrieve a transport plugin registered in a **DDSDomainParticipant** (p. 1139) by its alias.*

6.246.1 Detailed Description

*<<**interface**>> (p. 199)* The utility class used to configure RTI Connext pluggable transports.

6.246.2 Member Function Documentation

6.246.2.1 static NDDS_Transport_Handle_t NDDSTransportSupport::register_transport (DDSDomainParticipant * *participant_in*, NDDS_Transport_Plugin * *transport_in*, const DDS_StringSeq & *aliases_in*, const NDDS_Transport_Address_t & *network_address_in*) [static]

Register a transport plugin for use with a **DDSDomainParticipant** (p. 1139), assigning it a network_address.

A transport plugin instance can be used by exactly one **DDSDomainParticipant** (p. 1139) at a time.

When a DataWriter/DataReader is created, only those transports already registered to the corresponding **DDSDomainParticipant** (p. 1139) are available to the DataWriter/DataReader.

Builtin transports can be automatically registered by RTI Connext as a convenience to the user. See **Built-in Transport Plugins** (p. 136) for details on how to control the builtin transports that are automatically registered.

Precondition:

A disabled **DDSDomainParticipant** (p. 1139) and a transport plugin that will be registered exclusively with it.

Parameters:

participant_in *<<**in**>> (p. 200)* A non-null, disabled **DDSDomainParticipant** (p. 1139).

transport_in *<<**in**>> (p. 200)* A non-null transport plugin that is currently not registered with another **DDSDomainParticipant** (p. 1139).

aliases_in *<<**in**>> (p. 200)* A non-null sequence of strings used as aliases to symbolically refer to the transport plugins. The transport plugin

will be "available for use" by a **DDSEntity** (p. 1253) in the **DDSDomainParticipant** (p. 1139) if the transport alias list associated with the **DDSEntity** (p. 1253) contains one of these transport aliases. An empty alias list represents a wildcard and matches all aliases. Alias names for the builtin transports are defined in **TRANSPORT_BUILTIN** (p. 396).

network_address_in *<<in>> (p. 200)* The network address at which to register this transport plugin. The least significant `transport_in.property.address_bit_count` will be truncated. The remaining bits are the network address of the transport plugin. (see **Transport Class Attributes** (p. 126)).

Returns:

Upon success, a valid non-NIL transport handle, representing the association between the **DDSDomainParticipant** (p. 1139) and the transport plugin; a **NDDS_TRANSPORT_HANDLE_NIL** (p. 134) upon failure.

Note that a transport plugin's class name is automatically registered as an implicit alias for the plugin. Thus, a class name can be used to refer to all the transport plugin instance of that class.

See also:

Transport Class Attributes (p. 126)
Transport Network Address (p. 128)
Locator Format (p. 389)
NDDS_DISCOVERY_PEERS (p. 388)

6.246.2.2 static NDDS_Transport_Handle_t NDDSTransportSupport::lookup_transport (DDSDomainParticipant * *participant_in*, DDS_StringSeq & *aliases_out*, NDDS_Transport_Address_t & *network_address_out*, NDDS_Transport_Plugin * *transport_in*) [static]

Look up a transport plugin within a **DDSDomainParticipant** (p. 1139).

The transport plugin should have already been registered with the **DDSDomainParticipant** (p. 1139).

Parameters:

participant_in *<<in>> (p. 200)* A non-null **DDSDomainParticipant** (p. 1139).

aliases_out *<<inout>>* *(p. 200)* A sequence of string where the aliases used to refer to the transport plugin symbolically will be returned. null if not interested.

network_address_out *<<inout>>* *(p. 200)* The network address at which to register the transport plugin will be returned here. null if not interested.

transport_in *<<in>>* *(p. 200)* A non-null transport plugin that is already registered with the **DDSDomainParticipant** (p. 1139).

Returns:

Upon success, a valid non-NIL transport handle, representating the association between the **DDSDomainParticipant** (p. 1139) and the transport plugin; a **NDDS_TRANSPORT_HANDLE_NIL** (p. 134) upon failure.

See also:

Transport Class Attributes (p. 126)
Transport Network Address (p. 128)

6.246.2.3 static DDS_ReturnCode_t NDDSTransportSupport::add_send_route (const NDDS_Transport_Handle_t & *transport_handle_in*, const NDDS_Transport_Address_t & *address_range_in*, DDS_Long *address_range_bit_count_in*) [static]

Add a route for outgoing messages.

This method can be used to narrow the range of addresses to which outgoing messages can be sent.

Precondition:

A disabled **DDSDomainParticipant** (p. 1139).

Parameters:

transport_handle_in *<<in>>* *(p. 200)* A valid non-NIL transport handle as a result of a call to **NDDSTransportSupport::register_transport()** (p. 1560).

address_range_in *<<in>>* *(p. 200)* The outgoing address range for which to use this transport plugin.

address_range_bit_count_in *<<in>>* *(p. 200)* The number of most significant bits used to specify the address range.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_-PRECONDITION_NOT_MET** (p. 315).

See also:

Transport Send Route (p. 128)

6.246.2.4 static DDS_ReturnCode_t NDDSTransportSupport::add_receive_route (const NDDS_Transport_Handle_t & *transport_handle_in*, const NDDS_Transport_Address_t & *address_range_in*, DDS_Long *address_range_bit_count_in*) [static]

Add a route for incoming messages.

This method can be used to narrow the range of addresses at which to receive incoming messages.

Precondition:

A disabled **DDSDomainParticipant** (p. 1139).

Parameters:

transport_handle_in *<<in>> (p. 200)* A valid non-NIL transport handle as a result of a call to **NDDSTransportSupport::register_-transport()** (p. 1560).

address_range_in *<<in>> (p. 200)* The incoming address range for which to use this transport plugin.

address_range_bit_count_in *<<in>> (p. 200)* The number of most significant bits used to specify the address range.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_-PRECONDITION_NOT_MET** (p. 315).

See also:

Transport Receive Route (p. 129)

6.246.2.5 static DDS_ReturnCode_t NDDSTransportSupport::get_builtin_transport_property (DDSDomainParticipant * *participant_in*, DDS_TransportBuiltinKind *builtin_transport_kind_in*, struct NDDS_Transport_Property_t & *builtin_transport_property_inout*) [static]

Get the properties used to create a builtin transport plugin.

Retrieves the properties that will be used to create a builtin transport plugin.

Precondition:

The `builtin_transport_property_inout` parameter must be of the type specified by the `builtin_transport_kind_in`.

Parameters:

participant_in *<<in>>* *(p. 200)* A valid non-null **DDSDomainParticipant** (p. 1139)

builtin_transport_kind_in *<<in>>* *(p. 200)* The builtin transport kind for which to retrieve the properties.

builtin_transport_property_inout *<<inout>>* *(p. 200)* The storage area where the retrieved property will be output. The specific type required by the `builtin_transport_kind_in` must be used.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

See also:

NDDSTransportSupport::set_builtin_transport_property() (p. 1564)

6.246.2.6 static DDS_ReturnCode_t NDDSTransportSupport::set_builtin_transport_property (DDSDomainParticipant * *participant_in*, DDS_TransportBuiltinKind *builtin_transport_kind_in*, const struct NDDS_Transport_Property_t & *builtin_transport_property_in*) [static]

Set the properties used to create a builtin transport plugin.

Specifies the properties that will be used to create a builtin transport plugin.

If the builtin transport is already registered when this operation is called, these property changes will *not* have any effect. Builtin transport properties should

always be set before the transport is registered. See **Built-in Transport Plugins** (p. 136) for details on when a builtin transport is registered.

Precondition:

A disabled **DDSDomainParticipant** (p. 1139). The `builtin_transport_property_inout` parameter must be of the type specified by the `builtin_transport_kind_in`.

Parameters:

participant_in *<<in>> (p. 200)* A valid non-null **DDSDomainParticipant** (p. 1139) that has not been enabled.

builtin_transport_kind_in *<<in>> (p. 200)* The builtin transport kind for which to specificy the properties.

builtin_transport_property_in *<<inout>> (p. 200)* The new transport property that will be used to the create the builtin transport plugin. The specific type required by the `builtin_transport_kind_in` must be used.

Returns:

One of the **Standard Return Codes** (p. 314), or **DDS_RETCODE_PRECONDITION_NOT_MET** (p. 315).

See also:

NDDSTransportSupport::get_builtin_transport_property() (p. 1564)

6.246.2.7 static NDDS_Transport_Plugin* NDDSTransportSupport::get_transport_plugin (DDSDomainParticipant * *participant_in*, const char * *alias_in*) [static]

Retrieve a transport plugin registered in a **DDSDomainParticipant** (p. 1139) by its alias.

This method can be used to get a pointer to a transport Plugin that has been registered into the **DDSDomainParticipant** (p. 1139).

Parameters:

participant_in *<<in>> (p. 200)* A non-null **DDSDomainParticipant** (p. 1139).

alias_in *<<in>> (p. 200)* A non-null string used to symbolically refer to the transport plugins.

Returns:

Upon success, a valid non-null pointer to a registered plugin; a null pointer if a plugin with that alias is not registered/found in that participant.

6.247 NDDSUtility Class Reference

Unsupported utility APIs.

Static Public Member Functions

- static void **sleep** (const struct **DDS_Duration_t** &durationIn)

  Block the calling thread for the specified duration.

6.247.1 Detailed Description

Unsupported utility APIs.

Unsupported APIs used in example code. The static methods supplied by this class are used by the example code distributed with RTI Connext and generated by nddsgen. These methods are not supported by RTI.

6.247.2 Member Function Documentation

6.247.2.1 static void NDDSUtility::sleep (const struct DDS_Duration_t & *durationIn*) [static]

Block the calling thread for the specified duration.

Note that the achievable resolution of sleep is OS-dependent. That is, do not assume that you can sleep for 1 nanosecond just because you can specify a 1-nanosecond sleep duration via the API. The sleep resolution on most operating systems is usually 10 ms or greater.

Parameters:

durationIn *<<in>>* *(*p. 200*)* Sleep duration.

MT Safety:

safe

Examples:

HelloWorld_publisher.cxx, and **HelloWorld_subscriber.cxx**.

6.248 TransportAllocationSettings_t Struct Reference

Allocation settings used by various internal buffers.

6.248.1 Detailed Description

Allocation settings used by various internal buffers.

An allocation setting structure defines the rules of memory management used by internal buffers.

An internal buffer can provide blocks of memory of fixed size. They are used in several places of any transport, and this structure defines its starting size, limits, and how to increase its capacity.

It contains three values:

- initial_count: the number of individual elements that are allocated when the buffer is created.
- max_count: the maximum number of elements the buffer can hold. The buffer will grow up to this amount. After this limit is reached, new allocation requests will fail. For unlimited size, use the value NDDS_TRANSPORT_ALLOCATION_SETTINGS_MAX_COUNT_UNLIMITED
- incremental_count: The amount of elements that are allocated at every increment. You can use the value: NDDS_TRANSPORT_ALLOCATION_SETTINGS_INCREMENTAL_COUNT_AUTOMATIC to have the buffer double its size at every reallocation request.

Chapter 7

Example Documentation

7.1 HelloWorld.cxx

7.1.1 Programming Language Type Description

The following programming language specific type representation is generated by **rtiddsgen** (p. 220) for use in application code, where:

- **Foo** (p. 1443) = HelloWorld
- **FooSeq** (p. 1494) = HelloWorldSeq

7.1.1.1 HelloWorld.h

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorld.h]

```
/*
  WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

  This file was generated from HelloWorld.idl using "rtiddsgen".
  The rtiddsgen tool is part of the RTI Connext distribution.
  For more information, type 'rtiddsgen -help' at a command shell
  or consult the RTI Connext manual.
*/

#ifndef HelloWorld_1436885487_h
#define HelloWorld_1436885487_h

#ifndef NDDS_STANDALONE_TYPE
    #ifdef __cplusplus
        #ifndef ndds_cpp_h
            #include "ndds/ndds_cpp.h"
```

```
        #endif
    #else
        #ifndef ndds_c_h
            #include "ndds/ndds_c.h"
        #endif
    #endif
#else
    #include "ndds_standalone_type.h"
#endif


#ifdef __cplusplus
extern "C" {
#endif


extern const char *HelloWorldTYPENAME;


#ifdef __cplusplus
}
#endif


#ifdef __cplusplus
    struct HelloWorldSeq;

#ifndef NDDS_STANDALONE_TYPE
    class HelloWorldTypeSupport;
    class HelloWorldDataWriter;
    class HelloWorldDataReader;
#endif

#endif



class HelloWorld
{
public:
#ifdef __cplusplus
    typedef struct HelloWorldSeq Seq;

#ifndef NDDS_STANDALONE_TYPE
    typedef HelloWorldTypeSupport TypeSupport;
    typedef HelloWorldDataWriter DataWriter;
    typedef HelloWorldDataReader DataReader;
#endif

#endif

    char*  msg; /* maximum length = (128) */


};
```

```
#if (defined(RTI_WIN32) || defined (RTI_WINCE)) && defined(NDDS_USER_DLL_EXPORT)
  /* If the code is building on Windows, start exporting symbols.
   */
  #undef NDDSUSERDllExport
  #define NDDSUSERDllExport __declspec(dllexport)
#endif


NDDSUSERDllExport DDS_TypeCode* HelloWorld_get_typecode(void); /* Type code */


DDS_SEQUENCE(HelloWorldSeq, HelloWorld);

NDDSUSERDllExport
RTIBool HelloWorld_initialize(
        HelloWorld* self);

NDDSUSERDllExport
RTIBool HelloWorld_initialize_ex(
        HelloWorld* self,RTIBool allocatePointers,RTIBool allocateMemory);

NDDSUSERDllExport
void HelloWorld_finalize(
        HelloWorld* self);

NDDSUSERDllExport
void HelloWorld_finalize_ex(
        HelloWorld* self,RTIBool deletePointers);

NDDSUSERDllExport
RTIBool HelloWorld_copy(
        HelloWorld* dst,
        const HelloWorld* src);

#if (defined(RTI_WIN32) || defined (RTI_WINCE)) && defined(NDDS_USER_DLL_EXPORT)
  /* If the code is building on Windows, stop exporting symbols.
   */
  #undef NDDSUSERDllExport
  #define NDDSUSERDllExport
#endif



#endif /* HelloWorld_1436885487_h */
```

7.1.1.2 HelloWorld.cxx

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorld.cxx]

```
/*
  WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

  This file was generated from HelloWorld.idl using "rtiddsgen".
  The rtiddsgen tool is part of the RTI Connext distribution.
  For more information, type 'rtiddsgen -help' at a command shell
```

```
  or consult the RTI Connext manual.
*/


#ifndef NDDS_STANDALONE_TYPE
    #ifdef __cplusplus
        #ifndef ndds_cpp_h
            #include "ndds/ndds_cpp.h"
        #endif
        #ifndef dds_c_log_impl_h
            #include "dds_c/dds_c_log_impl.h"
        #endif
    #else
        #ifndef ndds_c_h
            #include "ndds/ndds_c.h"
        #endif
    #endif

    #ifndef cdr_type_h
        #include "cdr/cdr_type.h"
    #endif

    #ifndef osapi_heap_h
        #include "osapi/osapi_heap.h"
    #endif
#else
    #include "ndds_standalone_type.h"
#endif



#include "HelloWorld.h"

/* ========================================================================= */
const char *HelloWorldTYPENAME = "HelloWorld";

DDS_TypeCode* HelloWorld_get_typecode()
{
    static RTIBool is_initialized = RTI_FALSE;

    static DDS_TypeCode HelloWorld_g_tc_msg_string = DDS_INITIALIZE_STRING_TYPECODE(128);

    static DDS_TypeCode_Member HelloWorld_g_tc_members[1]=
    {
        {
            (char *)"msg",/* Member name */
            {
                0,/* Representation ID */
                DDS_BOOLEAN_FALSE,/* Is a pointer? */
                -1, /* Bitfield bits */
                NULL/* Member type code is assigned later */
            },
            0, /* Ignored */
            0, /* Ignored */
            0, /* Ignored */
            NULL, /* Ignored */
            DDS_BOOLEAN_FALSE, /* Is a key? */
```

```
            DDS_PRIVATE_MEMBER,/* Ignored */
            0,/* Ignored */
            NULL/* Ignored */
        }
    };

    static DDS_TypeCode HelloWorld_g_tc =
    {{
        DDS_TK_STRUCT,/* Kind */
        DDS_BOOLEAN_FALSE, /* Ignored */
        -1,/* Ignored */
        (char *)"HelloWorld", /* Name */
        NULL, /* Ignored */
        0, /* Ignored */
        0, /* Ignored */
        NULL, /* Ignored */
        1, /* Number of members */
        HelloWorld_g_tc_members, /* Members */
        DDS_VM_NONE /* Ignored */
    }}; /* Type code for HelloWorld*/

    if (is_initialized) {
        return &HelloWorld_g_tc;
    }


    HelloWorld_g_tc_members[0]._representation._typeCode = (RTICdrTypeCode *)&HelloWorld_g_tc_msg_string;

    is_initialized = RTI_TRUE;

    return &HelloWorld_g_tc;
}


RTIBool HelloWorld_initialize(
    HelloWorld* sample) {
  return HelloWorld_initialize_ex(sample,RTI_TRUE,RTI_TRUE);
}

RTIBool HelloWorld_initialize_ex(
    HelloWorld* sample,RTIBool allocatePointers,RTIBool allocateMemory)
{


    if (allocatePointers) {} /* To avoid warnings */
    if (allocateMemory) {} /* To avoid warnings */

    if (allocateMemory) {
        sample->msg = DDS_String_alloc((128));
        if (sample->msg == NULL) {
            return RTI_FALSE;
        }
    } else {
        if (sample->msg != NULL) {
            sample->msg[0] = '\0';
        }
    }
```

```
    return RTI_TRUE;
}

void HelloWorld_finalize(
    HelloWorld* sample)
{
    HelloWorld_finalize_ex(sample,RTI_TRUE);
}

void HelloWorld_finalize_ex(
    HelloWorld* sample,RTIBool deletePointers)
{
    if (sample) { } /* To avoid warnings */
    if (deletePointers) {} /* To avoid warnings */


    DDS_String_free(sample->msg);


}

RTIBool HelloWorld_copy(
    HelloWorld* dst,
    const HelloWorld* src)
{

    if (!RTICdrType_copyString(
        dst->msg, src->msg, (128) + 1)) {
        return RTI_FALSE;
    }



    return RTI_TRUE;
}


#define T HelloWorld
#define TSeq HelloWorldSeq
#define T_initialize_ex HelloWorld_initialize_ex
#define T_finalize_ex   HelloWorld_finalize_ex
#define T_copy       HelloWorld_copy

#ifndef NDDS_STANDALONE_TYPE
#include "dds_c/generic/dds_c_sequence_TSeq.gen"
#ifdef __cplusplus
#include "dds_cpp/generic/dds_cpp_sequence_TSeq.gen"
#endif
#else
#include "dds_c_sequence_TSeq.gen"
#ifdef __cplusplus
#include "dds_cpp_sequence_TSeq.gen"
#endif
#endif
```

```
#undef T_copy
#undef T_finalize_ex
#undef T_initialize_ex
#undef TSeq
#undef T
```

7.2 HelloWorld.idl

7.2.1 IDL Type Description

The data type to be disseminated by RTI Connext is described in language independent IDL. The IDL file is input to **rtiddsgen** (p. 220), which produces the following files.

The programming language specific type representation of the type **Foo** (p. 1443) = HelloWorld, for use in the application code.

- HelloWorld.h
- HelloWorld.cxx

User Data Type Support (p. 51) types as required by the DDS specification for use in the application code.

- HelloWorldSupport.h
- HelloWorldSupport.cxx

Methods required by the RTI Connext implementation. These contains the auto-generated methods for serializing and deserializing the type.

- HelloWorldPlugin.h
- HelloWorldPlugin.cxx

7.2.1.1 HelloWorld.idl

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorld.idl]

```
struct HelloWorld {
    string<128> msg;
};
```

7.3 HelloWorld_publisher.cxx

7.3.1 RTI Connext Publication Example

The publication example generated by **rtiddsgen** (p. 220). The example has been modified slightly to update the sample value.

7.3.1.1 HelloWorld_publisher.cxx

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorld_publisher.cxx]

```
/* HelloWorld_publisher.cxx

   A publication of data of type HelloWorld

   This file is derived from code automatically generated by the rtiddsgen
   command:

   rtiddsgen -language C++ -example <arch> HelloWorld.idl

   Example publication of type HelloWorld automatically generated by
   'rtiddsgen'. To test them follow these steps:

   (1) Compile this file and the example subscription.

   (2) Start the subscription on the same domain used for RTI Connext with the command
       objs/<arch>/HelloWorld_subscriber <domain_id> <sample_count>

   (3) Start the publication on the same domain used for RTI Connext with the command
       objs/<arch>/HelloWorld_publisher <domain_id> <sample_count>

   (4) [Optional] Specify the list of discovery initial peers and
       multicast receive addresses via an environment variable or a file
       (in the current working directory) called NDDS_DISCOVERY_PEERS.

   You can run any number of publishers and subscribers programs, and can
   add and remove them dynamically from the domain.


   Example:

       To run the example application on domain <domain_id>:

       On Unix:

       objs/<arch>/HelloWorld_publisher <domain_id>
       objs/<arch>/HelloWorld_subscriber <domain_id>

       On Windows:

       objs\<arch>\HelloWorld_publisher <domain_id>
       objs\<arch>\HelloWorld_subscriber <domain_id>
```

```
modification history
------------ -------
*/

#include <stdio.h>
#include <stdlib.h>
#include "ndds/ndds_cpp.h"
#include "HelloWorld.h"
#include "HelloWorldSupport.h"

/* Delete all entities */
static int publisher_shutdown(
    DDSDomainParticipant *participant)
{
    DDS_ReturnCode_t retcode;
    int status = 0;

    if (participant != NULL) {
        retcode = participant->delete_contained_entities();
        if (retcode != DDS_RETCODE_OK) {
            printf("delete_contained_entities error %d\n", retcode);
            status = -1;
        }

        retcode = DDSTheParticipantFactory->delete_participant(participant);
        if (retcode != DDS_RETCODE_OK) {
            printf("delete_participant error %d\n", retcode);
            status = -1;
        }
    }

    /* RTI Connext provides finalize_instance() method on
       domain participant factory for people who want to release memory used
       by the participant factory singleton. Uncomment the following block of
       code for clean destruction of the singleton. */
/*
    retcode = DDSDomainParticipantFactory::finalize_instance();
    if (retcode != DDS_RETCODE_OK) {
        printf("finalize_instance error %d\n", retcode);
        status = -1;
    }
*/
    return status;
}

extern "C" int publisher_main(int domainId, int sample_count)
{
    DDSDomainParticipant *participant = NULL;
    DDSPublisher *publisher = NULL;
    DDSTopic *topic = NULL;
    DDSDataWriter *writer = NULL;
    HelloWorldDataWriter *HelloWorld_writer = NULL;
    HelloWorld *instance = NULL;
    DDS_ReturnCode_t retcode;
    DDS_InstanceHandle_t instance_handle = DDS_HANDLE_NIL;
    const char *type_name = NULL;
```

```
int count = 0;
struct DDS_Duration_t send_period = {4,0};

/* To customize participant QoS, use
   DDSTheParticipantFactory->get_default_participant_qos()
   instead */
participant = DDSTheParticipantFactory->create_participant(
    domainId, DDS_PARTICIPANT_QOS_DEFAULT,
    NULL /* listener */, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    printf("create_participant error\n");
    publisher_shutdown(participant);
    return -1;
}

/* To customize publisher QoS, use
   participant->get_default_publisher_qos() instead */
publisher = participant->create_publisher(
    DDS_PUBLISHER_QOS_DEFAULT, NULL /* listener */, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    printf("create_publisher error\n");
    publisher_shutdown(participant);
    return -1;
}

/* Register type before creating topic */
type_name = HelloWorldTypeSupport::get_type_name();
retcode = HelloWorldTypeSupport::register_type(
    participant, type_name);
if (retcode != DDS_RETCODE_OK) {
    printf("register_type error %d\n", retcode);
    publisher_shutdown(participant);
    return -1;
}

/* To customize topic QoS, use
   participant->get_default_topic_qos() instead */
topic = participant->create_topic(
    "Example HelloWorld",
    type_name, DDS_TOPIC_QOS_DEFAULT, NULL /* listener */,
    DDS_STATUS_MASK_NONE);
if (topic == NULL) {
    printf("create_topic error\n");
    publisher_shutdown(participant);
    return -1;
}

/* To customize data writer QoS, use
   publisher->get_default_datawriter_qos() instead */
writer = publisher->create_datawriter(
    topic, DDS_DATAWRITER_QOS_DEFAULT, NULL /* listener */,
    DDS_STATUS_MASK_NONE);
if (writer == NULL) {
    printf("create_datawriter error\n");
    publisher_shutdown(participant);
    return -1;
}
```

```
    HelloWorld_writer = HelloWorldDataWriter::narrow(writer);
    if (HelloWorld_writer == NULL) {
        printf("DataWriter narrow error\n");
        publisher_shutdown(participant);
        return -1;
    }

    /* Create data sample for writing */
    instance = HelloWorldTypeSupport::create_data();
    if (instance == NULL) {
        printf("HelloWorldTypeSupport::create_data error\n");
        publisher_shutdown(participant);
        return -1;
    }

    /* For data type that has key, if the same instance is going to be
       written multiple times, initialize the key here
       and register the keyed instance prior to writing */
/*
    instance_handle = HelloWorld_writer->register_instance(*instance);
*/

    /* Main loop */
    for (count=0; (sample_count == 0) || (count < sample_count); ++count) {

        printf("Writing HelloWorld, count %d\n", count);

        /* Modify the data to be sent here */
        sprintf(instance->msg, "Hello World! (%d)", count);

        retcode = HelloWorld_writer->write(*instance, instance_handle);
        if (retcode != DDS_RETCODE_OK) {
            printf("write error %d\n", retcode);
        }

        NDDSUtility::sleep(send_period);
    }

/*
    retcode = HelloWorld_writer->unregister_instance(
        *instance, instance_handle);
    if (retcode != DDS_RETCODE_OK) {
        printf("unregister instance error %d\n", retcode);
    }
*/

    /* Delete data sample */
    retcode = HelloWorldTypeSupport::delete_data(instance);
    if (retcode != DDS_RETCODE_OK) {
        printf("HelloWorldTypeSupport::delete_data error %d\n", retcode);
    }

    /* Delete all entities */
    return publisher_shutdown(participant);
}

#if defined(RTI_WINCE)
```

```
int wmain(int argc, wchar_t** argv)
{
    int domainId = 0;
    int sample_count = 0; /* infinite loop */

    if (argc >= 2) {
        domainId = _wtoi(argv[1]);
    }
    if (argc >= 3) {
        sample_count = _wtoi(argv[2]);
    }

    /* Uncomment this to turn on additional logging
    NDDSConfigLogger::get_instance()->
        set_verbosity_by_category(NDDS_CONFIG_LOG_CATEGORY_API,
                                  NDDS_CONFIG_LOG_VERBOSITY_STATUS_ALL);
    */

    return publisher_main(domainId, sample_count);
}

#elif !(defined(RTI_VXWORKS) && !defined(__RTP__)) && !defined(RTI_PSOS)
int main(int argc, char *argv[])
{
    int domainId = 0;
    int sample_count = 0; /* infinite loop */

    if (argc >= 2) {
        domainId = atoi(argv[1]);
    }
    if (argc >= 3) {
        sample_count = atoi(argv[2]);
    }

    /* Uncomment this to turn on additional logging
    NDDSConfigLogger::get_instance()->
        set_verbosity_by_category(NDDS_CONFIG_LOG_CATEGORY_API,
                                  NDDS_CONFIG_LOG_VERBOSITY_STATUS_ALL);
    */

    return publisher_main(domainId, sample_count);
}
#endif
```

7.4 HelloWorld_subscriber.cxx

7.4.1 RTI Connext Subscription Example

The unmodified subscription example generated by **rtiddsgen** (p. 220).

7.4.1.1 HelloWorld_subscriber.cxx

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorld_subscriber.cxx]

```
/* HelloWorld_subscriber.cxx

   A subscription example

   This file is derived from code automatically generated by the rtiddsgen
   command:

   rtiddsgen -language C++ -example <arch> HelloWorld.idl

   Example subscription of type HelloWorld automatically generated by
   'rtiddsgen'. To test them follow these steps:

   (1) Compile this file and the example publication.

   (2) Start the subscription with the command
       objs/<arch>/HelloWorld_subscriber <domain_id> <sample_count>

   (3) Start the publication with the command
       objs/<arch>/HelloWorld_publisher <domain_id> <sample_count>

   (4) [Optional] Specify the list of discovery initial peers and
       multicast receive addresses via an environment variable or a file
       (in the current working directory) called NDDS_DISCOVERY_PEERS.

   You can run any number of publishers and subscribers programs, and can
   add and remove them dynamically from the domain.


   Example:

       To run the example application on domain <domain_id>:

       On Unix:

       objs/<arch>/HelloWorld_publisher <domain_id>
       objs/<arch>/HelloWorld_subscriber <domain_id>

       On Windows:

       objs\<arch>\HelloWorld_publisher <domain_id>
       objs\<arch>\HelloWorld_subscriber <domain_id>


modification history
```

```
------------ -------
*/

#include <stdio.h>
#include <stdlib.h>
#ifdef RTI_VX653
#include <vThreadsData.h>
#endif
#include "HelloWorld.h"
#include "HelloWorldSupport.h"
#include "ndds/ndds_cpp.h"

class HelloWorldListener : public DDSDataReaderListener {
  public:
    virtual void on_requested_deadline_missed(
        DDSDataReader* /*reader*/,
        const DDS_RequestedDeadlineMissedStatus& /*status*/) {}

    virtual void on_requested_incompatible_qos(
        DDSDataReader* /*reader*/,
        const DDS_RequestedIncompatibleQosStatus& /*status*/) {}

    virtual void on_sample_rejected(
        DDSDataReader* /*reader*/,
        const DDS_SampleRejectedStatus& /*status*/) {}

    virtual void on_liveliness_changed(
        DDSDataReader* /*reader*/,
        const DDS_LivelinessChangedStatus& /*status*/) {}

    virtual void on_sample_lost(
        DDSDataReader* /*reader*/,
        const DDS_SampleLostStatus& /*status*/) {}

    virtual void on_subscription_matched(
        DDSDataReader* /*reader*/,
        const DDS_SubscriptionMatchedStatus& /*status*/) {}

    virtual void on_data_available(DDSDataReader* reader);
};

void HelloWorldListener::on_data_available(DDSDataReader* reader)
{
    HelloWorldDataReader *HelloWorld_reader = NULL;
    HelloWorldSeq data_seq;
    DDS_SampleInfoSeq info_seq;
    DDS_ReturnCode_t retcode;
    int i;

    HelloWorld_reader = HelloWorldDataReader::narrow(reader);
    if (HelloWorld_reader == NULL) {
        printf("DataReader narrow error\n");
        return;
    }

    retcode = HelloWorld_reader->take(
        data_seq, info_seq, DDS_LENGTH_UNLIMITED,
```

```
        DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);

    if (retcode == DDS_RETCODE_NO_DATA) {
        return;
    } else if (retcode != DDS_RETCODE_OK) {
        printf("take error %d\n", retcode);
        return;
    }

    for (i = 0; i < data_seq.length(); ++i) {
        if (info_seq[i].valid_data) {
            HelloWorldTypeSupport::print_data(&data_seq[i]);
        }
    }

    retcode = HelloWorld_reader->return_loan(data_seq, info_seq);
    if (retcode != DDS_RETCODE_OK) {
        printf("return loan error %d\n", retcode);
    }
}

/* Delete all entities */
static int subscriber_shutdown(
    DDSDomainParticipant *participant)
{
    DDS_ReturnCode_t retcode;
    int status = 0;

    if (participant != NULL) {
        retcode = participant->delete_contained_entities();
        if (retcode != DDS_RETCODE_OK) {
            printf("delete_contained_entities error %d\n", retcode);
            status = -1;
        }

        retcode = DDSTheParticipantFactory->delete_participant(participant);
        if (retcode != DDS_RETCODE_OK) {
            printf("delete_participant error %d\n", retcode);
            status = -1;
        }
    }

    /* RTI Connext provides the finalize_instance() method on
       domain participant factory for people who want to release memory used
       by the participant factory. Uncomment the following block of code for
       clean destruction of the singleton. */
/*
    retcode = DDSDomainParticipantFactory::finalize_instance();
    if (retcode != DDS_RETCODE_OK) {
        printf("finalize_instance error %d\n", retcode);
        status = -1;
    }
*/
    return status;
}

extern "C" int subscriber_main(int domainId, int sample_count)
```

```
{
    DDSDomainParticipant *participant = NULL;
    DDSSubscriber *subscriber = NULL;
    DDSTopic *topic = NULL;
    HelloWorldListener *reader_listener = NULL;
    DDSDataReader *reader = NULL;
    DDS_ReturnCode_t retcode;
    const char *type_name = NULL;
    int count = 0;
    DDS_Duration_t receive_period = {4,0};
    int status = 0;

    /* To customize the participant QoS, use
       DDSTheParticipantFactory->get_default_participant_qos() */
    participant = DDSTheParticipantFactory->create_participant(
        domainId, DDS_PARTICIPANT_QOS_DEFAULT,
        NULL /* listener */, DDS_STATUS_MASK_NONE);
    if (participant == NULL) {
        printf("create_participant error\n");
        subscriber_shutdown(participant);
        return -1;
    }

    /* To customize the subscriber QoS, use
       participant->get_default_subscriber_qos() */
    subscriber = participant->create_subscriber(
        DDS_SUBSCRIBER_QOS_DEFAULT, NULL /* listener */, DDS_STATUS_MASK_NONE);
    if (subscriber == NULL) {
        printf("create_subscriber error\n");
        subscriber_shutdown(participant);
        return -1;
    }

    /* Register the type before creating the topic */
    type_name = HelloWorldTypeSupport::get_type_name();
    retcode = HelloWorldTypeSupport::register_type(
        participant, type_name);
    if (retcode != DDS_RETCODE_OK) {
        printf("register_type error %d\n", retcode);
        subscriber_shutdown(participant);
        return -1;
    }

    /* To customize the topic QoS, use
       participant->get_default_topic_qos() */
    topic = participant->create_topic(
        "Example HelloWorld",
        type_name, DDS_TOPIC_QOS_DEFAULT, NULL /* listener */,
        DDS_STATUS_MASK_NONE);
    if (topic == NULL) {
        printf("create_topic error\n");
        subscriber_shutdown(participant);
        return -1;
    }

    /* Create a data reader listener */
    reader_listener = new HelloWorldListener();
```

```
    /* To customize the data reader QoS, use
       subscriber->get_default_datareader_qos() */
    reader = subscriber->create_datareader(
        topic, DDS_DATAREADER_QOS_DEFAULT, reader_listener,
        DDS_STATUS_MASK_ALL);
    if (reader == NULL) {
        printf("create_datareader error\n");
        subscriber_shutdown(participant);
        delete reader_listener;
        return -1;
    }

    /* Main loop */
    for (count=0; (sample_count == 0) || (count < sample_count); ++count) {

        printf("HelloWorld subscriber sleeping for %d sec...\n",
               receive_period.sec);

        NDDSUtility::sleep(receive_period);
    }

    /* Delete all entities */
    status = subscriber_shutdown(participant);
    delete reader_listener;

    return status;
}

#if defined(RTI_WINCE)
int wmain(int argc, wchar_t** argv)
{
    int domainId = 0;
    int sample_count = 0; /* infinite loop */

    if (argc >= 2) {
        domainId = _wtoi(argv[1]);
    }
    if (argc >= 3) {
        sample_count = _wtoi(argv[2]);
    }

    /* Uncomment this to turn on additional logging
    NDDSConfigLogger::get_instance()->
        set_verbosity_by_category(NDDS_CONFIG_LOG_CATEGORY_API,
                                  NDDS_CONFIG_LOG_VERBOSITY_STATUS_ALL);
    */

    return subscriber_main(domainId, sample_count);
}

#elif !(defined(RTI_VXWORKS) && !defined(__RTP__)) && !defined(RTI_PSOS)
int main(int argc, char *argv[])
{
    int domainId = 0;
    int sample_count = 0; /* infinite loop */
```

```
    if (argc >= 2) {
        domainId = atoi(argv[1]);
    }
    if (argc >= 3) {
        sample_count = atoi(argv[2]);
    }


    /* Uncomment this to turn on additional logging
    NDDSConfigLogger::get_instance()->
        set_verbosity_by_category(NDDS_CONFIG_LOG_CATEGORY_API,
                                  NDDS_CONFIG_LOG_VERBOSITY_STATUS_ALL);
    */

    return subscriber_main(domainId, sample_count);
}
#endif

#ifdef RTI_VX653
const unsigned char* __ctype = *(__ctypePtrGet());

extern "C" void usrAppInit ()
{
#ifdef  USER_APPL_INIT
    USER_APPL_INIT;         /* for backwards compatibility */
#endif

    /* add application specific code here */
    taskSpawn("sub", RTI_OSAPI_THREAD_PRIORITY_NORMAL, 0x8, 0x150000, (FUNCPTR)subscriber_main, 0, 0, 0, 0, 0, 0, 0, 0,

}
#endif
```

7.5 HelloWorldPlugin.cxx

7.5.1 RTI Connext Implementation Support

Files generated by **rtiddsgen** (p. 220) that provided methods for type specific serialization and deserialization, to support the RTI Connext implementation.

7.5.1.1 HelloWorldPlugin.h

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorldPlugin.h]

```
/*
  WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

  This file was generated from HelloWorld.idl using "rtiddsgen".
  The rtiddsgen tool is part of the RTI Connext distribution.
  For more information, type 'rtiddsgen -help' at a command shell
  or consult the RTI Connext manual.
*/

#ifndef HelloWorldPlugin_1436885487_h
#define HelloWorldPlugin_1436885487_h

#include "HelloWorld.h"



struct RTICdrStream;

#ifndef pres_typePlugin_h
#include "pres/pres_typePlugin.h"
#endif


#if (defined(RTI_WIN32) || defined (RTI_WINCE)) && defined(NDDS_USER_DLL_EXPORT)
/* If the code is building on Windows, start exporting symbols.
*/
#undef NDDSUSERDllExport
#define NDDSUSERDllExport __declspec(dllexport)
#endif


#ifdef __cplusplus
extern "C" {
#endif

#define HelloWorld_LAST_MEMBER_ID 0

#define HelloWorldPlugin_get_sample PRESTypePluginDefaultEndpointData_getSample
#define HelloWorldPlugin_return_sample PRESTypePluginDefaultEndpointData_returnSample
#define HelloWorldPlugin_get_buffer PRESTypePluginDefaultEndpointData_getBuffer
#define HelloWorldPlugin_return_buffer PRESTypePluginDefaultEndpointData_returnBuffer
```

```
#define HelloWorldPlugin_create_sample PRESTypePluginDefaultEndpointData_createSample
#define HelloWorldPlugin_destroy_sample PRESTypePluginDefaultEndpointData_deleteSample

/* ------------------------------------------------------------------------------------
    Support functions:
 * ------------------------------------------------------------------------------------ */

NDDSUSERDllExport extern HelloWorld*
HelloWorldPluginSupport_create_data_ex(RTIBool allocate_pointers);

NDDSUSERDllExport extern HelloWorld*
HelloWorldPluginSupport_create_data(void);

NDDSUSERDllExport extern RTIBool
HelloWorldPluginSupport_copy_data(
    HelloWorld *out,
    const HelloWorld *in);

NDDSUSERDllExport extern void
HelloWorldPluginSupport_destroy_data_ex(
    HelloWorld *sample,RTIBool deallocate_pointers);

NDDSUSERDllExport extern void
HelloWorldPluginSupport_destroy_data(
    HelloWorld *sample);

NDDSUSERDllExport extern void
HelloWorldPluginSupport_print_data(
    const HelloWorld *sample,
    const char *desc,
    unsigned int indent);


/* ------------------------------------------------------------------------
    Callback functions:
 * ------------------------------------------------------------------------ */

NDDSUSERDllExport extern PRESTypePluginParticipantData
HelloWorldPlugin_on_participant_attached(
    void *registration_data,
    const struct PRESTypePluginParticipantInfo *participant_info,
    RTIBool top_level_registration,
    void *container_plugin_context,
    RTICdrTypeCode *typeCode);

NDDSUSERDllExport extern void
HelloWorldPlugin_on_participant_detached(
    PRESTypePluginParticipantData participant_data);

NDDSUSERDllExport extern PRESTypePluginEndpointData
HelloWorldPlugin_on_endpoint_attached(
    PRESTypePluginParticipantData participant_data,
    const struct PRESTypePluginEndpointInfo *endpoint_info,
    RTIBool top_level_registration,
    void *container_plugin_context);
```

```
NDDSUSERDllExport extern void
HelloWorldPlugin_on_endpoint_detached(
    PRESTypePluginEndpointData endpoint_data);


NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_copy_sample(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *out,
    const HelloWorld *in);

/* ----------------------------------------------------------------------------
    (De)Serialize functions:
 * ------------------------------------------------------------------------- */

NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_serialize(
    PRESTypePluginEndpointData endpoint_data,
    const HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool serialize_encapsulation,
    RTIEncapsulationId encapsulation_id,
    RTIBool serialize_sample,
    void *endpoint_plugin_qos);

NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_deserialize_sample(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_sample,
    void *endpoint_plugin_qos);


NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_deserialize(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld **sample,
    RTIBool * drop_sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_sample,
    void *endpoint_plugin_qos);




NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_skip(
    PRESTypePluginEndpointData endpoint_data,
    struct RTICdrStream *stream,
    RTIBool skip_encapsulation,
    RTIBool skip_sample,
    void *endpoint_plugin_qos);
```

```
NDDSUSERDllExport extern unsigned int
HelloWorldPlugin_get_serialized_sample_max_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment);

NDDSUSERDllExport extern unsigned int
HelloWorldPlugin_get_serialized_sample_min_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment);

NDDSUSERDllExport extern unsigned int
HelloWorldPlugin_get_serialized_sample_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment,
    const HelloWorld * sample);



/* ------------------------------------------------------------------------------
    Key Management functions:
 * ------------------------------------------------------------------------------ */

NDDSUSERDllExport extern PRESTypePluginKeyKind
HelloWorldPlugin_get_key_kind(void);

NDDSUSERDllExport extern unsigned int
HelloWorldPlugin_get_serialized_key_max_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment);

NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_serialize_key(
    PRESTypePluginEndpointData endpoint_data,
    const HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool serialize_encapsulation,
    RTIEncapsulationId encapsulation_id,
    RTIBool serialize_key,
    void *endpoint_plugin_qos);

NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_deserialize_key_sample(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld * sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_key,
    void *endpoint_plugin_qos);
```

```
NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_deserialize_key(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld ** sample,
    RTIBool * drop_sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_key,
    void *endpoint_plugin_qos);


NDDSUSERDllExport extern RTIBool
HelloWorldPlugin_serialized_sample_to_key(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_key,
    void *endpoint_plugin_qos);


/* Plugin Functions */
NDDSUSERDllExport extern struct PRESTypePlugin*
HelloWorldPlugin_new(void);

NDDSUSERDllExport extern void
HelloWorldPlugin_delete(struct PRESTypePlugin *);

#ifdef __cplusplus
}
#endif


#if (defined(RTI_WIN32) || defined (RTI_WINCE)) && defined(NDDS_USER_DLL_EXPORT)
/* If the code is building on Windows, stop exporting symbols.
*/
#undef NDDSUSERDllExport
#define NDDSUSERDllExport
#endif

#endif /* HelloWorldPlugin_1436885487_h */
```

7.5.1.2 HelloWorldPlugin.cxx

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorldPlugin.cxx]

```
/*
  WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

  This file was generated from HelloWorld.idl using "rtiddsgen".
  The rtiddsgen tool is part of the RTI Connext distribution.
  For more information, type 'rtiddsgen -help' at a command shell
  or consult the RTI Connext manual.
*/
```

```
#include <string.h>

#ifdef __cplusplus
#ifndef ndds_cpp_h
  #include "ndds/ndds_cpp.h"
#endif
#else
#ifndef ndds_c_h
  #include "ndds/ndds_c.h"
#endif
#endif

#ifndef osapi_type_h
  #include "osapi/osapi_type.h"
#endif
#ifndef osapi_heap_h
  #include "osapi/osapi_heap.h"
#endif

#ifndef osapi_utility_h
  #include "osapi/osapi_utility.h"
#endif

#ifndef cdr_type_h
  #include "cdr/cdr_type.h"
#endif

#ifndef cdr_type_object_h
  #include "cdr/cdr_typeObject.h"
#endif

#ifndef cdr_encapsulation_h
  #include "cdr/cdr_encapsulation.h"
#endif

#ifndef cdr_stream_h
  #include "cdr/cdr_stream.h"
#endif

#ifndef pres_typePlugin_h
  #include "pres/pres_typePlugin.h"
#endif



#include "HelloWorldPlugin.h"


/* ------------------------------------------------------------------------
 *  Type HelloWorld
 * ------------------------------------------------------------------------ */

/* ------------------------------------------------------------------------
    Support functions:
 * ------------------------------------------------------------------------ */
```

```
HelloWorld *
HelloWorldPluginSupport_create_data_ex(RTIBool allocate_pointers){
    HelloWorld *sample = NULL;

    RTIOsapiHeap_allocateStructure(
        &sample, HelloWorld);

    if(sample != NULL) {
        if (!HelloWorld_initialize_ex(sample,allocate_pointers, RTI_TRUE)) {
            RTIOsapiHeap_freeStructure(sample);
            return NULL;
        }
    }
    return sample;
}


HelloWorld *
HelloWorldPluginSupport_create_data(void)
{
    return HelloWorldPluginSupport_create_data_ex(RTI_TRUE);
}


void
HelloWorldPluginSupport_destroy_data_ex(
    HelloWorld *sample,RTIBool deallocate_pointers) {

    HelloWorld_finalize_ex(sample,deallocate_pointers);

    RTIOsapiHeap_freeStructure(sample);
}


void
HelloWorldPluginSupport_destroy_data(
    HelloWorld *sample) {

    HelloWorldPluginSupport_destroy_data_ex(sample,RTI_TRUE);

}


RTIBool
HelloWorldPluginSupport_copy_data(
    HelloWorld *dst,
    const HelloWorld *src)
{
    return HelloWorld_copy(dst,src);
}


void
HelloWorldPluginSupport_print_data(
    const HelloWorld *sample,
    const char *desc,
```

```
    unsigned int indent_level)
{


    RTICdrType_printIndent(indent_level);

    if (desc != NULL) {
      RTILog_debug("%s:\n", desc);
    } else {
      RTILog_debug("\n");
    }

    if (sample == NULL) {
      RTILog_debug("NULL\n");
      return;
    }


    if (&sample->msg==NULL) {
        RTICdrType_printString(
            NULL, "msg", indent_level + 1);
    } else {
        RTICdrType_printString(
            sample->msg, "msg", indent_level + 1);
    }



}


/* ------------------------------------------------------------------------
    Callback functions:
 * ------------------------------------------------------------------------ */



PRESTypePluginParticipantData
HelloWorldPlugin_on_participant_attached(
    void *registration_data,
    const struct PRESTypePluginParticipantInfo *participant_info,
    RTIBool top_level_registration,
    void *container_plugin_context,
    RTICdrTypeCode *type_code)
{

    if (registration_data) {} /* To avoid warnings */
    if (participant_info) {} /* To avoid warnings */
    if (top_level_registration) {} /* To avoid warnings */
    if (container_plugin_context) {} /* To avoid warnings */
    if (type_code) {} /* To avoid warnings */
    return PRESTypePluginDefaultParticipantData_new(participant_info);

}


void
```

```
HelloWorldPlugin_on_participant_detached(
    PRESTypePluginParticipantData participant_data)
{

  PRESTypePluginDefaultParticipantData_delete(participant_data);
}


PRESTypePluginEndpointData
HelloWorldPlugin_on_endpoint_attached(
    PRESTypePluginParticipantData participant_data,
    const struct PRESTypePluginEndpointInfo *endpoint_info,
    RTIBool top_level_registration,
    void *containerPluginContext)
{
    PRESTypePluginEndpointData epd = NULL;

    unsigned int serializedSampleMaxSize;

   if (top_level_registration) {} /* To avoid warnings */
   if (containerPluginContext) {} /* To avoid warnings */

    epd = PRESTypePluginDefaultEndpointData_new(
            participant_data,
            endpoint_info,
            (PRESTypePluginDefaultEndpointDataCreateSampleFunction)
            HelloWorldPluginSupport_create_data,
            (PRESTypePluginDefaultEndpointDataDestroySampleFunction)
            HelloWorldPluginSupport_destroy_data,
            NULL, NULL);

    if (epd == NULL) {
        return NULL;
    }



    if (endpoint_info->endpointKind == PRES_TYPEPLUGIN_ENDPOINT_WRITER) {
        serializedSampleMaxSize = HelloWorldPlugin_get_serialized_sample_max_size(
            epd,RTI_FALSE,RTI_CDR_ENCAPSULATION_ID_CDR_BE,0);

        PRESTypePluginDefaultEndpointData_setMaxSizeSerializedSample(epd, serializedSampleMaxSize);

        if (PRESTypePluginDefaultEndpointData_createWriterPool(
                epd,
                endpoint_info,
            (PRESTypePluginGetSerializedSampleMaxSizeFunction)
                HelloWorldPlugin_get_serialized_sample_max_size, epd,
            (PRESTypePluginGetSerializedSampleSizeFunction)
            HelloWorldPlugin_get_serialized_sample_size,
            epd) == RTI_FALSE) {
            PRESTypePluginDefaultEndpointData_delete(epd);
            return NULL;
        }
    }
```

```
    return epd;
}


void
HelloWorldPlugin_on_endpoint_detached(
    PRESTypePluginEndpointData endpoint_data)
{

    PRESTypePluginDefaultEndpointData_delete(endpoint_data);
}



RTIBool
HelloWorldPlugin_copy_sample(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *dst,
    const HelloWorld *src)
{
    if (endpoint_data) {} /* To avoid warnings */
    return HelloWorldPluginSupport_copy_data(dst,src);
}

/* ------------------------------------------------------------------------
    (De)Serialize functions:
 * ------------------------------------------------------------------------ */

unsigned int
HelloWorldPlugin_get_serialized_sample_max_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment);


RTIBool
HelloWorldPlugin_serialize(
    PRESTypePluginEndpointData endpoint_data,
    const HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool serialize_encapsulation,
    RTIEncapsulationId encapsulation_id,
    RTIBool serialize_sample,
    void *endpoint_plugin_qos)
{
    char * position = NULL;
    RTIBool retval = RTI_TRUE;

    if (endpoint_data) {} /* To avoid warnings */
    if (endpoint_plugin_qos) {} /* To avoid warnings */


    if(serialize_encapsulation) {

        if (!RTICdrStream_serializeAndSetCdrEncapsulation(stream, encapsulation_id)) {
```

```
            return RTI_FALSE;
        }

        position = RTICdrStream_resetAlignment(stream);

    }


    if(serialize_sample) {

    if (!RTICdrStream_serializeString(
        stream, sample->msg, (128) + 1)) {
        return RTI_FALSE;
    }


    }


    if(serialize_encapsulation) {
        RTICdrStream_restoreAlignment(stream,position);
    }


  return retval;
}


RTIBool
HelloWorldPlugin_deserialize_sample(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_sample,
    void *endpoint_plugin_qos)
{
    char * position = NULL;

    RTIBool done = RTI_FALSE;

    if (endpoint_data) {} /* To avoid warnings */
    if (endpoint_plugin_qos) {} /* To avoid warnings */


    if(deserialize_encapsulation) {
        /* Deserialize encapsulation */
        if (!RTICdrStream_deserializeAndSetCdrEncapsulation(stream)) {
            return RTI_FALSE;
        }

        position = RTICdrStream_resetAlignment(stream);

    }


    if(deserialize_sample) {
```

```
        HelloWorld_initialize_ex(sample, RTI_FALSE, RTI_FALSE);

    if (!RTICdrStream_deserializeString(
        stream, sample->msg, (128) + 1)) {
        goto fin;
    }


    }

    done = RTI_TRUE;
fin:
    if (done != RTI_TRUE && RTICdrStream_getRemainder(stream) >  0) {
        return RTI_FALSE;
    }

    if(deserialize_encapsulation) {
        RTICdrStream_restoreAlignment(stream,position);
    }


    return RTI_TRUE;
}



RTIBool
HelloWorldPlugin_deserialize(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld **sample,
    RTIBool * drop_sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_sample,
    void *endpoint_plugin_qos)
{

    if (drop_sample) {} /* To avoid warnings */

    return HelloWorldPlugin_deserialize_sample(
        endpoint_data, (sample != NULL)?*sample:NULL,
        stream, deserialize_encapsulation, deserialize_sample,
        endpoint_plugin_qos);

}



RTIBool HelloWorldPlugin_skip(
    PRESTypePluginEndpointData endpoint_data,
    struct RTICdrStream *stream,
    RTIBool skip_encapsulation,
    RTIBool skip_sample,
    void *endpoint_plugin_qos)
{
```

```
    char * position = NULL;

    RTIBool done = RTI_FALSE;

    if (endpoint_data) {} /* To avoid warnings */
    if (endpoint_plugin_qos) {} /* To avoid warnings */


    if(skip_encapsulation) {
        if (!RTICdrStream_skipEncapsulation(stream)) {
            return RTI_FALSE;
        }


        position = RTICdrStream_resetAlignment(stream);

    }

    if (skip_sample) {

    if (!RTICdrStream_skipString(stream, (128) + 1)) {
        goto fin;
    }



    }


    done = RTI_TRUE;
fin:
    if (done != RTI_TRUE && RTICdrStream_getRemainder(stream) >  0) {
        return RTI_FALSE;
    }

    if(skip_encapsulation) {
        RTICdrStream_restoreAlignment(stream,position);
    }


    return RTI_TRUE;
}


unsigned int
HelloWorldPlugin_get_serialized_sample_max_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment)
{

    unsigned int initial_alignment = current_alignment;

    unsigned int encapsulation_size = current_alignment;

    if (endpoint_data) {} /* To avoid warnings */
```

```
    if (include_encapsulation) {

        if (!RTICdrEncapsulation_validEncapsulationId(encapsulation_id)) {
            return 1;
        }

        RTICdrStream_getEncapsulationSize(encapsulation_size);
        encapsulation_size -= current_alignment;
        current_alignment = 0;
        initial_alignment = 0;

    }


    current_alignment +=  RTICdrType_getStringMaxSizeSerialized(
        current_alignment, (128) + 1);


    if (include_encapsulation) {
        current_alignment += encapsulation_size;
    }

    return current_alignment - initial_alignment;
}


unsigned int
HelloWorldPlugin_get_serialized_sample_min_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment)
{

    unsigned int initial_alignment = current_alignment;

    unsigned int encapsulation_size = current_alignment;

    if (endpoint_data) {} /* To avoid warnings */


    if (include_encapsulation) {

        if (!RTICdrEncapsulation_validEncapsulationId(encapsulation_id)) {
            return 1;
        }

        RTICdrStream_getEncapsulationSize(encapsulation_size);
        encapsulation_size -= current_alignment;
        current_alignment = 0;
        initial_alignment = 0;

    }
```

```
    current_alignment +=  RTICdrType_getStringMaxSizeSerialized(
        current_alignment, 1);


    if (include_encapsulation) {
        current_alignment += encapsulation_size;
    }

    return current_alignment - initial_alignment;
}


/* Returns the size of the sample in its serialized form (in bytes).
 * It can also be an estimation in excess of the real buffer needed
 * during a call to the serialize() function.
 * The value reported does not have to include the space for the
 * encapsulation flags.
 */
unsigned int
HelloWorldPlugin_get_serialized_sample_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment,
    const HelloWorld * sample)
{

    unsigned int initial_alignment = current_alignment;

    unsigned int encapsulation_size = current_alignment;

    if (endpoint_data) {} /* To avoid warnings */
    if (sample) {} /* To avoid warnings */


    if (include_encapsulation) {

        if (!RTICdrEncapsulation_validEncapsulationId(encapsulation_id)) {
            return 1;
        }

        RTICdrStream_getEncapsulationSize(encapsulation_size);
        encapsulation_size -= current_alignment;
        current_alignment = 0;
        initial_alignment = 0;

    }


    current_alignment += RTICdrType_getStringSerializedSize(
        current_alignment, sample->msg);


    if (include_encapsulation) {
        current_alignment += encapsulation_size;
    }
```

```
    return current_alignment - initial_alignment;
}




/* ------------------------------------------------------------------------
    Key Management functions:
 * ------------------------------------------------------------------------ */


PRESTypePluginKeyKind
HelloWorldPlugin_get_key_kind(void)
{

    return PRES_TYPEPLUGIN_NO_KEY;

}


RTIBool
HelloWorldPlugin_serialize_key(
    PRESTypePluginEndpointData endpoint_data,
    const HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool serialize_encapsulation,
    RTIEncapsulationId encapsulation_id,
    RTIBool serialize_key,
    void *endpoint_plugin_qos)
{
    char * position = NULL;

    if (endpoint_data) {} /* To avoid warnings */
    if (endpoint_plugin_qos) {} /* To avoid warnings */


    if(serialize_encapsulation) {

        if (!RTICdrStream_serializeAndSetCdrEncapsulation(stream, encapsulation_id)) {
            return RTI_FALSE;
        }


        position = RTICdrStream_resetAlignment(stream);

    }

    if(serialize_key) {

        if (!HelloWorldPlugin_serialize(
                endpoint_data,
                sample,
                stream,
                RTI_FALSE, encapsulation_id,
```

```
                RTI_TRUE,
                endpoint_plugin_qos)) {
            return RTI_FALSE;
        }

    }


    if(serialize_encapsulation) {
        RTICdrStream_restoreAlignment(stream,position);
    }


    return RTI_TRUE;
}


RTIBool HelloWorldPlugin_deserialize_key_sample(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_key,
    void *endpoint_plugin_qos)
{
    char * position = NULL;

    if (endpoint_data) {} /* To avoid warnings */
    if (endpoint_plugin_qos) {} /* To avoid warnings */


    if(deserialize_encapsulation) {
        /* Deserialize encapsulation */
        if (!RTICdrStream_deserializeAndSetCdrEncapsulation(stream)) {
            return RTI_FALSE;
        }


        position = RTICdrStream_resetAlignment(stream);

    }

    if (deserialize_key) {

        if (!HelloWorldPlugin_deserialize_sample(
                endpoint_data, sample, stream,
                RTI_FALSE, RTI_TRUE,
                endpoint_plugin_qos)) {
            return RTI_FALSE;
        }

    }


    if(deserialize_encapsulation) {
        RTICdrStream_restoreAlignment(stream,position);
    }
```

```
    return RTI_TRUE;
}



RTIBool HelloWorldPlugin_deserialize_key(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld **sample,
    RTIBool * drop_sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_key,
    void *endpoint_plugin_qos)
{
    if (drop_sample) {} /* To avoid warnings */
    return HelloWorldPlugin_deserialize_key_sample(
        endpoint_data, (sample != NULL)?*sample:NULL, stream,
        deserialize_encapsulation, deserialize_key, endpoint_plugin_qos);
}



unsigned int
HelloWorldPlugin_get_serialized_key_max_size(
    PRESTypePluginEndpointData endpoint_data,
    RTIBool include_encapsulation,
    RTIEncapsulationId encapsulation_id,
    unsigned int current_alignment)
{

    unsigned int encapsulation_size = current_alignment;


    unsigned int initial_alignment = current_alignment;


    if (endpoint_data) {} /* To avoid warnings */


    if (include_encapsulation) {
        if (!RTICdrEncapsulation_validEncapsulationId(encapsulation_id)) {
            return 1;
        }


        RTICdrStream_getEncapsulationSize(encapsulation_size);
        encapsulation_size -= current_alignment;
        current_alignment = 0;
        initial_alignment = 0;

    }


    current_alignment += HelloWorldPlugin_get_serialized_sample_max_size(
        endpoint_data,RTI_FALSE, encapsulation_id, current_alignment);
```

```
    if (include_encapsulation) {
        current_alignment += encapsulation_size;
    }

    return current_alignment - initial_alignment;
}


RTIBool
HelloWorldPlugin_serialized_sample_to_key(
    PRESTypePluginEndpointData endpoint_data,
    HelloWorld *sample,
    struct RTICdrStream *stream,
    RTIBool deserialize_encapsulation,
    RTIBool deserialize_key,
    void *endpoint_plugin_qos)
{
    char * position = NULL;

    RTIBool done = RTI_FALSE;

    if (stream == NULL) goto fin; /* To avoid warnings */


    if(deserialize_encapsulation) {
        if (!RTICdrStream_deserializeAndSetCdrEncapsulation(stream)) {
            return RTI_FALSE;
        }

        position = RTICdrStream_resetAlignment(stream);

    }

    if (deserialize_key) {

        if (!HelloWorldPlugin_deserialize_sample(
            endpoint_data, sample, stream, RTI_FALSE,
            RTI_TRUE, endpoint_plugin_qos)) {
            return RTI_FALSE;
        }

    }


    done = RTI_TRUE;
fin:
    if (done != RTI_TRUE && RTICdrStream_getRemainder(stream) >  0) {
        return RTI_FALSE;
    }

    if(deserialize_encapsulation) {
        RTICdrStream_restoreAlignment(stream,position);
    }


    return RTI_TRUE;
```

```
}



/* ----------------------------------------------------------------------
 * Plug-in Installation Methods
 * ---------------------------------------------------------------------- */

struct PRESTypePlugin *HelloWorldPlugin_new(void)
{
    struct PRESTypePlugin *plugin = NULL;
    const struct PRESTypePluginVersion PLUGIN_VERSION =
        PRES_TYPE_PLUGIN_VERSION_2_0;

    RTIOsapiHeap_allocateStructure(
        &plugin, struct PRESTypePlugin);
    if (plugin == NULL) {
       return NULL;
    }

    plugin->version = PLUGIN_VERSION;

    /* set up parent's function pointers */
    plugin->onParticipantAttached =
        (PRESTypePluginOnParticipantAttachedCallback)
        HelloWorldPlugin_on_participant_attached;
    plugin->onParticipantDetached =
        (PRESTypePluginOnParticipantDetachedCallback)
        HelloWorldPlugin_on_participant_detached;
    plugin->onEndpointAttached =
        (PRESTypePluginOnEndpointAttachedCallback)
        HelloWorldPlugin_on_endpoint_attached;
    plugin->onEndpointDetached =
        (PRESTypePluginOnEndpointDetachedCallback)
        HelloWorldPlugin_on_endpoint_detached;

    plugin->copySampleFnc =
        (PRESTypePluginCopySampleFunction)
        HelloWorldPlugin_copy_sample;
    plugin->createSampleFnc =
        (PRESTypePluginCreateSampleFunction)
        HelloWorldPlugin_create_sample;
    plugin->destroySampleFnc =
        (PRESTypePluginDestroySampleFunction)
        HelloWorldPlugin_destroy_sample;

    plugin->serializeFnc =
        (PRESTypePluginSerializeFunction)
        HelloWorldPlugin_serialize;
    plugin->deserializeFnc =
        (PRESTypePluginDeserializeFunction)
        HelloWorldPlugin_deserialize;
    plugin->getSerializedSampleMaxSizeFnc =
        (PRESTypePluginGetSerializedSampleMaxSizeFunction)
        HelloWorldPlugin_get_serialized_sample_max_size;
    plugin->getSerializedSampleMinSizeFnc =
```

```
        (PRESTypePluginGetSerializedSampleMinSizeFunction)
        HelloWorldPlugin_get_serialized_sample_min_size;


    plugin->getSampleFnc =
        (PRESTypePluginGetSampleFunction)
        HelloWorldPlugin_get_sample;
    plugin->returnSampleFnc =
        (PRESTypePluginReturnSampleFunction)
        HelloWorldPlugin_return_sample;

    plugin->getKeyKindFnc =
        (PRESTypePluginGetKeyKindFunction)
        HelloWorldPlugin_get_key_kind;


    /* These functions are only used for keyed types. As this is not a keyed
    type they are all set to NULL
    */
    plugin->serializeKeyFnc = NULL;
    plugin->deserializeKeyFnc = NULL;
    plugin->getKeyFnc = NULL;
    plugin->returnKeyFnc = NULL;
    plugin->instanceToKeyFnc = NULL;
    plugin->keyToInstanceFnc = NULL;
    plugin->getSerializedKeyMaxSizeFnc = NULL;
    plugin->instanceToKeyHashFnc = NULL;
    plugin->serializedSampleToKeyHashFnc = NULL;
    plugin->serializedKeyToKeyHashFnc = NULL;

    plugin->typeCode =  (struct RTICdrTypeCode *)HelloWorld_get_typecode();

    plugin->languageKind = PRES_TYPEPLUGIN_DDS_TYPE;

    /* Serialized buffer */
    plugin->getBuffer =
        (PRESTypePluginGetBufferFunction)
        HelloWorldPlugin_get_buffer;
    plugin->returnBuffer =
        (PRESTypePluginReturnBufferFunction)
        HelloWorldPlugin_return_buffer;
    plugin->getSerializedSampleSizeFnc =
        (PRESTypePluginGetSerializedSampleSizeFunction)
        HelloWorldPlugin_get_serialized_sample_size;

    plugin->endpointTypeName = HelloWorldTYPENAME;

    return plugin;
}

void
HelloWorldPlugin_delete(struct PRESTypePlugin *plugin)
{
    RTIOsapiHeap_freeStructure(plugin);
}
```

7.6 HelloWorldSupport.cxx

7.6.1 User Data Type Support

Files generated by **rtiddsgen** (p. 220) that implement the type specific APIs required by the DDS specification, as described in the **User Data Type Support** (p. 51), where:

- **FooTypeSupport** (p. 1509) = HelloWorldTypeSupport
- **FooDataWriter** (p. 1475) = HelloWorldDataWriter
- **FooDataReader** (p. 1444) = HelloWorldDataReader

7.6.1.1 HelloWorldSupport.h

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorldSupport.h]

```
/*
  WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

  This file was generated from HelloWorld.idl using "rtiddsgen".
  The rtiddsgen tool is part of the RTI Connext distribution.
  For more information, type 'rtiddsgen -help' at a command shell
  or consult the RTI Connext manual.
*/

#ifndef HelloWorldSupport_1436885487_h
#define HelloWorldSupport_1436885487_h

/* Uses */
#include "HelloWorld.h"



#ifdef __cplusplus
#ifndef ndds_cpp_h
  #include "ndds/ndds_cpp.h"
#endif
#else
#ifndef ndds_c_h
  #include "ndds/ndds_c.h"
#endif
#endif



/* ========================================================================= */
#if (defined(RTI_WIN32) || defined (RTI_WINCE)) && defined(NDDS_USER_DLL_EXPORT)
  /* If the code is building on Windows, start exporting symbols.
   */
  #undef NDDSUSERDllExport
```

```
  #define NDDSUSERDllExport __declspec(dllexport)

#ifdef __cplusplus
  /* If we're building on Windows, explicitly import the superclasses of
   * the types declared below.
   */
  class __declspec(dllimport) DDSTypeSupport;
  class __declspec(dllimport) DDSDataWriter;
  class __declspec(dllimport) DDSDataReader;
#endif

#endif

#ifdef __cplusplus

DDS_TYPESUPPORT_CPP(HelloWorldTypeSupport, HelloWorld);

DDS_DATAWRITER_CPP(HelloWorldDataWriter, HelloWorld);
DDS_DATAREADER_CPP(HelloWorldDataReader, HelloWorldSeq, HelloWorld);


#else

DDS_TYPESUPPORT_C(HelloWorldTypeSupport, HelloWorld);
DDS_DATAWRITER_C(HelloWorldDataWriter, HelloWorld);
DDS_DATAREADER_C(HelloWorldDataReader, HelloWorldSeq, HelloWorld);

#endif

#if (defined(RTI_WIN32) || defined (RTI_WINCE)) && defined(NDDS_USER_DLL_EXPORT)
  /* If the code is building on Windows, stop exporting symbols.
   */
  #undef NDDSUSERDllExport
  #define NDDSUSERDllExport
#endif



#endif  /* HelloWorldSupport_1436885487_h */
```

7.6.1.2 HelloWorldSupport.cxx

[$(NDDSHOME)/example/CPP/helloWorld/HelloWorldSupport.cxx]

```
/*
  WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

  This file was generated from HelloWorld.idl using "rtiddsgen".
  The rtiddsgen tool is part of the RTI Connext distribution.
  For more information, type 'rtiddsgen -help' at a command shell
  or consult the RTI Connext manual.
*/

#include "HelloWorldSupport.h"
#include "HelloWorldPlugin.h"
```

```
#ifdef __cplusplus
    #ifndef dds_c_log_impl_h
         #include "dds_c/dds_c_log_impl.h"
    #endif
#endif




/* ========================================================================= */
/* ------------------------------------------------------------------------- */
/* DDSDataWriter
*/

/* Requires */
#define TTYPENAME   HelloWorldTYPENAME

/* Defines */
#define TDataWriter HelloWorldDataWriter
#define TData       HelloWorld


#ifdef __cplusplus
#include "dds_cpp/generic/dds_cpp_data_TDataWriter.gen"
#else
#include "dds_c/generic/dds_c_data_TDataWriter.gen"
#endif


#undef TDataWriter
#undef TData

#undef TTYPENAME

/* ------------------------------------------------------------------------- */
/* DDSDataReader
*/

/* Requires */
#define TTYPENAME   HelloWorldTYPENAME

/* Defines */
#define TDataReader HelloWorldDataReader
#define TDataSeq    HelloWorldSeq
#define TData       HelloWorld


#ifdef __cplusplus
#include "dds_cpp/generic/dds_cpp_data_TDataReader.gen"
#else
#include "dds_c/generic/dds_c_data_TDataReader.gen"
#endif
```

```
#undef TDataReader
#undef TDataSeq
#undef TData

#undef TTYPENAME

/* ------------------------------------------------------------------ */
/* TypeSupport

  <<IMPLEMENTATION >>

   Requires:  TTYPENAME,
              TPlugin_new
              TPlugin_delete
   Defines:   TTypeSupport, TData, TDataReader, TDataWriter
*/

/* Requires */
#define TTYPENAME    HelloWorldTYPENAME
#define TPlugin_new  HelloWorldPlugin_new
#define TPlugin_delete  HelloWorldPlugin_delete

/* Defines */
#define TTypeSupport HelloWorldTypeSupport
#define TData        HelloWorld
#define TDataReader  HelloWorldDataReader
#define TDataWriter  HelloWorldDataWriter
#ifdef __cplusplus



#include "dds_cpp/generic/dds_cpp_data_TTypeSupport.gen"



#else
#include "dds_c/generic/dds_c_data_TTypeSupport.gen"
#endif
#undef TTypeSupport
#undef TData
#undef TDataReader
#undef TDataWriter

#undef TTYPENAME
#undef TPlugin_new
#undef TPlugin_delete
```