VORTEX
OPENSPLICE
Java Reference Guide
Release 6.x

Vortex OpenSplice

Java Reference Guide

PRISMTECH
AN ADLINK COMPANY

Part Number: OS-JREFG

Copyright Notice

CONTENTS

Table of Contents

PRISMTECH

PRISMTECH

>>> PRISMTECH

PRISMTECH

PRISMTECH

PRISMTECH

PRISMTECH

Table of Contents

PRISMTECH

List of Figures

List of Figures

PRISMTECH

Preface

About the Java Reference Guide

The *Java Reference Guide* provides details of the Vortex OpenSplice (*Subscription Paradigm for the Logical Interconnection of Concurrent Engines*) Application Programming Interfaces for the Java language.

This reference guide is based on the OMG's *Data Distribution Service Specification* and *Java Language Mapping Specification*.

The Java Reference Guide describes the Data Centric Publish Subscribe (DCPS) layer. The purpose of the DCPS is the distribution of data (publish/subscribe). The structure of the DCPS is divided into five modules. Each module consists of several classes, which in turn generally contain several operations.

Intended Audience

The *Java Reference Guide* is intended to be used by Java programmers who are using Vortex OpenSplice to develop applications.

Organisation

The *Java Reference Guide* is organised into the following topics.

The *Introduction* describes the details of the document structure.

Chapter 1, *DCPS API General Description*, is a general description of the DCPS API and its error codes.

Chapter 2, *DCPS Modules*, provides the detailed description of the DCPS modules.

Chapter 3, *DCPS Classes and Operations*, provides the detailed description of the DCPS classes, structs and operations.

The following appendices are included, as well as a *Bibliography* containing reference material and a *Glossary*:

Appendix A, *Quality Of Service*

Appendix B, *API Constants and Types*

Appendix C, *Platform Specific Model IDL Interface*

Appendix D, *SampleStates, ViewStates and InstanceStates*

Appendix E, *Interface Inheritance*

Appendix F, *Listeners, Conditions and Waitsets*

Appendix G, *Topic Definitions*

Appendix H, *DCPS Queries and Filters*

Appendix I, *Built-in Topics*

Conventions

The conventions listed below are used to guide and assist the reader in understanding the Java Reference Guide.

⚠   Item of special significance or where caution needs to be taken.

i   Item contains helpful hint or special information.

WIN   Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.

UNIX   Information applies to Unix based systems (*e.g.* Solaris) only.

C   C language specific

C++   C++ language specific

Java   Java language specific

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross references, *e.g. Contacts* on page xxv, are hypertext links: click on the reference to go to the item.

```
%   Commands or input which the user enters on the
    command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to "example" and kind field to an empty string
newName[0] = new NameComponent ("example", "");
```

Italics and ***Italic Bold*** are used to indicate new terms, or emphasise an item.

Sans-serif Bold is used to indicate user-related actions, *e.g.* **File > Save** from a menu.

Step 1:   One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters	**European Head Office**
PrismTech Corporation	PrismTech Limited
400 TradeCenter	PrismTech House
Suite 5900	5th Avenue Business Park
Woburn, MA	Gateshead
01801	NE11 0NG
USA	UK
Tel: +1 781 569 5819	Tel: +44 (0)191 497 9900
	Fax: +44 (0)191 497 9901

Web:	*http://www.prismtech.com*
Technical questions:	*crc@prismtech.com* (Customer Response Center)
Sales enquiries:	*sales@prismtech.com*

INTRODUCTION

About the Java Reference Guide

Document Structure

The Java Reference Guide document structure is based on the structure of the DCPS Platform Independent Model (DCPS PIM) of the Data Distribution Service Specification. The detailed description is subdivided into the PIM Modules, which are then subdivided into classes.

Some of the classes are implemented as structs in the DCPS Platform Specific Model (DCPS PSM) of the Data Distribution Service Specification, as indicated in the Interface Description Language (IDL) chapter of the PSM (see Appendix C, *Platform Specific Model IDL Interface*.

- In the classes as described in the PIM, which are implemented as a class in the PSM, the operations are described in detail.

- In the classes as described in the PIM, which are implemented as a struct in the PSM, the struct contents are described in detail.

- The order of the modules and classes is conform the PIM part.

- The order of the operations or struct contents is alphabetical.

- Each description of a class or struct starts with the API description header file.

DDS-DCPS
detailed description

Modules . . .

Structs . . .   **Classes . . .**
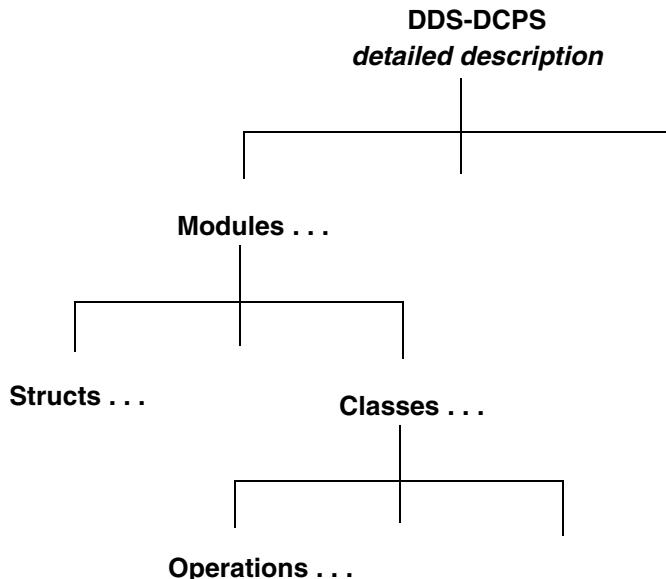
Operations . . .

Figure 1 Java Reference Guide Document Structure

Operations

Operations are described in detail in the class they are implemented in. This means inherited operations, only refer to the operation in the class they are inherited from. Abstract operations only refer to the type specific implementations in their respective derived class. An exception is made for the abstract operations (implemented as an interface), which must be implemented in the application. Abstract interfaces are described in detail in this manual. In the API description header file, the inherited and abstract operations are commented out since they are not implemented in this class.

PRISMTECH

API Reference

1 DCPS API General Description

The structure of the DCPS is divided into modules, which are described in detail in the next chapter. Each module consists of several classes, which in turn may contain several operations.

Some of these operations have an operation return code of type int, which possible value is defined in the next table. See Section 1.1, IDL Mapping to Java, on page 8 for an explanation of IDL constants.

Table 1 Return Codes

Return Code	Description
RETCODE_OK	Successful return
RETCODE_ERROR	Generic, unspecified error
RETCODE_BAD_PARAMETER	Illegal parameter value
RETCODE_UNSUPPORTED	Unsupported operation or `QosPolicy` setting. Can only be returned by operations that are optional or operations that uses an optional `<Entity>QoS` as a parameter
RETCODE_ALREADY_DELETED	The object target of this operation has already been deleted
RETCODE_OUT_OF_RESOURCES	Service ran out of the resources needed to complete the operation
RETCODE_NOT_ENABLED	Operation invoked on an `Entity` that is not yet enabled
RETCODE_IMMUTABLE_POLICY	Application attempted to modify an immutable `QosPolicy`
RETCODE_INCONSISTENT_POLICY	Application specified a set of policies that are not consistent with each other
RETCODE_PRECONDITION_NOT_MET	A pre-condition for the operation was not met

Table 1 Return Codes

Return Code	Description
RETCODE_TIMEOUT	The operation timed out
RETCODE_ILLEGAL_OPERATION	An operation was invoked on an inappropriate object or at an inappropriate time (as determined by QosPolicies that control the behaviour of the object in question). There is no precondition that could be changed to make the operation succeed. In Java, this code can never be returned!
RETCODE_NO_DATA	Indicates a situation where the operation did not return any data

Return Codes are located in the package DDS. The operation return codes RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_UNSUPPORTED and RETCODE_ALREADY_DELETED are default for operations that return an operation return code and are therefore not explicitly mentioned in the DDS specification. However, in this manual they are mentioned along with each operation.

Some operations are not implemented. These operations are mentioned including their synopsis, but not described in this manual and return RETCODE_UNSUPPORTED when called from the application. All constants and types are listed in Appendix B, *API Constants and Types*.

The return code RETCODE_ILLEGAL_OPERATION can never be returned in Java: it indicates that you try to invoke an operation on the wrong class, which in a language like Java, that natively supports Object Orientation by supporting the concept of classes with internal operations, is never possible.

1.1 IDL Mapping to Java

1.1.1 IDL-Constant Mapping

Constants, declared outside the scope of a IDL interface, are mapped to a public interface with the same name as the constant and containing a field, named value, that holds the constant's value. For example the IDL constant:

```
module DDS {
    const ReturnCode_t RETCODE_OK = 0;
}
```

is mapped to the following interface class:

```
Package DDS;
```

PRISMTECH

```
public interface RETCODE_OK {
    public static final int value = (int)(0);
}
```

The value of the constant can be retrieved with:

```
RETCODE_OK.value
```

A few examples of the usage of constants:

- use the *OR* operation to compose a value:

```
condition.set_enabled_statuses (
    INCONSISTENT_TOPIC_STATUS.value |
        SAMPLE_REJECTED_STATUS.value);
```

- use the *AND* operation to check a value:

```
if ((status & INCONSISTENT_TOPIC_STATUS.value) ==
        INCONSISTENT_TOPIC_STATUS.value) {
        error = true;
}
```

1.1.2 IDL-Sequence Mapping

An IDL sequence is mapped to a Java array with the same name. The holder class, see section 1.5 on page 11, for the sequence is generated too. For example the IDL sequence:

```
typedef sequence<long> Data
```

is mapped to the following holder class:

```
public final class DataHolder
{
    public int[] value = null;
    public DataHolder () { }
    public DataHolder (int[] initialValue)
    {
            value = initialValue;
    }
}
```

The array can be initiated using the constructor:

```
int[] data = new int[10];
DataHolder dataHolder = new DataHolder(data);
```

To reference to an element of the array use for example:

```
int x = dataHolder.value[2];
```

The length of the array is found using:

```
int length = dataHolder.value.length;
```

1.2 Thread Safety

All operations are thread safe.

1.3 Signal Handling

UNIX   Every application that participates in a domain should register signal-handlers in order to protect the data distribution service in case of an exception or termination request. This is done automatically when the application calls the `DDS.DomainParticipantFactory.get_instance()` operation. The data distribution service distinguishes between two kinds of signals: synchronous (*i.e.* exceptions) and asynchronous signals (*i.e.* termination requests).

1.3.1 Synchronous Signals

The data distribution service registers a signal-handler for the following synchronous signals: `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGFPE`, `SIGBUS`, `SIGSEGV` and `SIGSYS`. If a signal-handler is already registered for any of these signals it will be chained by the handlers registered by the data distribution service. Upon receiving any of the mentioned signals, the signal-handler will synchronously detach the application from the domain and call any chained handler if available. This allows core dumps to be created when an error occurs in application-code, without sacrificing the integrity of the data distribution service. Because the signal is processed synchronously, the offending thread will not be able to continue.

Synchronous signals can also be received asynchronously from another process (*i.e.* `'kill -ABRT <pid>'`). This is handled by the signal-handlers registered by the data distribution service and the behaviour will mimic the behaviour of a regular synchronous signal, occurring at the point in the application when the signal is received. A log message will be recorded stating that an asynchronously received synchronous signal occurred, including the source of the signal.

1.3.2 Asynchronous Signals

The asynchronous signal-handlers are only registered by the data distribution service if the application did not already register a handler, nor set the ignore-flag for these signals. If the data distribution service handlers are registered, the default handlers are chained. The signals that are handled are: `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGHUP` and `SIGPIPE`. When receiving any of these signals, the handlers of the data distribution service will ensure a disconnection from the domain. The default handler will in turn cause the application to terminate immediately.

1.3.3 Preloading the Java signal handler library

The Java Virtual Machine uses its own signal handlers internally. Which signals are caught by it, depends on the platform it is running on. The Splice-DDS signal handlers conflict with the ones used internally by the Java Virtual Machine. To avoid this problem the signal-chaining facility of the virtual machine must be used. The signal-chaining facility is installed by loading the shared library `libjsig.so` before `libc`, `libthead` and/or `libpthread`. The `libjsig.so` library is

provided with the Java virtual machine. This library ensures that calls such as `signal()`, `sigset()` and `sigaction()` are intercepted so that they do not actually replace the Java virtual machine signal handlers. Instead, signal handlers of the application are "chained" behind the handlers of the Java virtual machine. When any of these signals are raised and found not to be targeted at the Java virtual machine, the signals invoke the signal handlers installed by the application. When just the Data Distribution System is used without any application signal handlers, no action needs to be taken.

The library `libjsig.so` can be loaded before the Java application by using the `LD_PRELOAD` environment variable, for example:

```
LD_PRELOAD = <libjvm directory>/libjsig.so
export LD_PRELOAD
```

1.4 Memory Management

When objects are being created, they will occupy memory resources. Release of the memory resources is the responsibility of the Java garbage collector. The memory of an object is released, after all references to this object have run out of scope or have explicitly been removed (set to `null`).

1.5 Parameter Passing

Support for passing out and inout parameter in a function call requires the use of additional "holder" classes. These classes are available for all of the data types in the DDS package and are generated for all user-defined IDL types, except those defined by typedefs. For user-defined IDL types, the holder class name is constructed by appending "Holder" to the mapped Java name of the type, i.e. FooHolder for user-defined type Foo.

Each holder class has a default constructor and has a public instance member, named value, which is the typed value. The default constructor sets the value field to `null` for object references.

The Holder class for the type `DataReaderSeq` is shown below:

```
public final class DataReaderSeqHolder
{
    public DDS.DataReader value[] = null;
    public DataReaderSeqHolder ()
    {
    }
}
```

Example of usage of the DataReaderSeqHolder class:

```
DDS.DataReader reader;
DDS.DataReaderSeqHolder readersholder = new
DDS.DataReaderSeqHolder;
```

```
// Get data readers objects
subscriber.get_datareaders (readersholder, DDS.ANY_STATE.value,
DDS.ANY_VIEW.value,
    DDS.ANY_INSTANCE_STATE.value);
// Get first data reader object from the sequence
if(readersholder.value[0] != null)
    reader = readersholder.value[0];
```

In this example a holder is declared for the type DataReaderSeq. After the call to the function subscriber.get_datareaders, the readersholder contains a reference to an array of DataReader objects. A reader is retrieved in the last line of the example code.

The Holder classes defined in DCPS are listed in Table 2:

Table 2 DCPS Holder Classes

AllDataDisposedTopicStatusHolder	PartitionQosPolicyHolder
BuiltinTopicKey_tHolder	PresentationQosPolicyAccessScopeKindHolder
ConditionHolder	PresentationQosPolicyHolder
ConditionSeqHolder	PublicationBuiltinTopicDataHolder
ContentFilteredTopicHolder	PublicationMatchStatusHolder
DataReaderHolder	PublisherHolder
DataReaderListenerHolder	PublisherListenerHolder
DataReaderQosHolder	PublisherQosHolder
DataReaderSeqHolder	QosPolicyCountHolder
DataWriterHolder	QosPolicyCountSeqHolder
DataWriterListenerHolder	QueryConditionHolder
DataWriterQosHolder	ReadConditionHolder
DeadlineQosPolicyHolder	ReaderDataLifecycleQosPolicyHolder
DestinationOrderQosPolicyHolder	ReliabilityQosPolicyHolder
DestinationOrderQosPolicyKindHolder	ReliabilityQosPolicyKindHolder
DomainParticipantFactoryHolder	RequestedDeadlineMissedStatusHolder
DomainParticipantHolder	RequestedIncompatibleQosStatusHolder
DomainParticipantListenerHolder	ResourceLimitsQosPolicyHolder
DomainParticipantQosHolder	SampleInfoHolder
DurabilityQosPolicyHolder	SampleInfoSeqHolder
DurabilityQosPolicyKindHolder	SampleLostStatusHolder
Duration_tHolder	SampleRejectedStatusHolder
EntityFactoryQosPolicyHolder	SampleRejectedStatusKindHolder
EntityHolder	SampleStateSeqHolder

PRISMTECH

Table 2 DCPS Holder Classes (Continued)

ExtDomainParticipantListenerHolder	StatusConditionHolder
ExtTopicListenerHolder	StringSeqHolder
GroupDataQosPolicyHolder	SubscriberHolder
GuardConditionHolder	SubscriberListenerHolder
HistoryQosPolicyHolder	SubscriberQosHolder
HistoryQosPolicyKindHolder	SubscriptionBuiltinTopicDataHolder
InconsistentTopicStatusHolder	SubscriptionMatchStatusHolder
InstanceHandleSeqHolder	Time_tHolder
InstanceStateSeqHolder	TimeBasedFilterQosPolicyHolder
LatencyBudgetQosPolicyHolder	TopicBuiltinTopicDataHolder
LifespanQosPolicyHolder	TopicDataQosPolicyHolder
ListenerHolder	TopicDescriptionHolder
LivelinessChangedStatusHolder	TopicHolder
LivelinessLostStatusHolder	TopicListenerHolder
LivelinessQosPolicyHolder	TopicQosHolder
LivelinessQosPolicyKindHolder	TopicSeqHolder
MultiTopicHolder	TransportPriorityQosPolicyHolder
OfferedDeadlineMissedStatusHolder	TypeSupportHolder
OfferedIncompatibleQosStatusHolder	UserDataQosPolicyHolder
OwnershipQosPolicyHolder	ViewStateSeqHolder
OwnershipQosPolicyKindHolder	WaitSetHolder
OwnershipStrengthQosPolicyHolder	WriterDataLifecycleQosPolicyHolder
ParticipantBuiltinTopicDataHolder	

1.6 Casting of Objects

Instances of classes can also be cast to instances of other classes, with one restriction: the class of the object you're casting and the class you're casting it to must be related by inheritance. So a class can be cast to a subclass, not to any random class.

For any user-defined interface a Helper class is generated with the method narrow() to cast to the derived class type. The following Java code is generated:

```
abstract public class <typename> Helper {
    public static <typename> narrow(java.lang.Object obj);
}
```

The static method narrow(), defined in the Helper class, is a method to cast a superclass object to the <typename> object.

For the user-defined structure Foo, the following Helper classes are generated:

- FooDataWriterHelper
- FooDataReaderHelper
- FooTypeSupportHelper

The example below demonstrates the casting of an DataReader type to a FooDataReader type by calling the function narrow:

```
// Create the DomainParticipant participant first
...
// The participant creates a new subscriber
DDS.Subscriber subscriber =
    participant.create_subscriber(subqos, sublistener);
// The subscriber creates a new FooDataReader
DS.DataReader reader = subscriber.create_datareader(topic,
    drqos, drlistener);
// The DataReader is cast to a FooDataReader,
// using the static method narrow() in the Helper class
FooDataReader foo_reader =
    FooDataReaderHelper.narrow(reader);
```

1.7 Listeners Interfaces

The Listener provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a QosPolicy setting, etc. The Listener is related to changes in communication status.

The Listener interfaces are designed as an interface at PIM level. In other words, such an interface is part of the application which must implement the interface operations. A user-defined class for these operations must be provided by the application which must extend from the specific Listener class. All Listener operations must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

Each DCPS Entity supports its own specialized kind of Listener. Therefore, the following Listeners are available:

- DomainParticipantListener
- ExtDomainParticipantListener
- TopicListener
- ExtTopicListener
- PublisherListener
- DataWriterListener

PRISMTECH

- `SubscriberListener`
- `DataReaderListener`

Example

For example, since a `DataReader` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` must be of type `DataReaderListener`. This interface must be implemented by the application. A user-defined class must be provided by the application (for instance `My_DataReaderListener`) which must extend from the `DataReaderListener` class. *All* `DataReaderListener` operations *must* be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

As an example, one of the operations in the `DataReaderListener` is the `on_liveliness_changed`. This operation (implemented by the application) will be called by the Data Distribution Service when the liveliness of the associated `DataWriter` has changed. In other words, it serves as a callback function to the event of a change in liveliness. The parameters of the operation are supplied by the Data Distribution Service. In this example, the reference to the `DataReader` and the status of the liveliness are provided.

1.8 Inheritance of Abstract Operations

This manual is based on:

- PIM part of the DDS-DCPS specification (to describe the modules);
- PSM part of the DDS-DCPS specification (to describe the classes and operations).

At PIM level, inheritance is used to define abstract classes and operations. The OMG IDL PSM (see Appendix C, *Platform Specific Model IDL Interface*) defines the interface for an application to interact with the Data Distribution Service. The DCPS API for the Java programming language conforms to the *OMG Java Language Mapping Specification* (Java to IDL).

Inheritance of operations is not implemented when different type parameters for the same operation are used. In this case operations are implemented in their respective derived class (e.g. `get_qos` and `set_qos`). These operations are commented out in the IDL PSM.

1.9 Shutdown hook

In Vortex OpenSplice DCPS, both the stand-alone and CORBA-cohabited Java APIs will by default install a shutdown hook in every process that uses the API.

The hook is installed when the singleton `DomainParticipantFactory` is obtained for the first time using the method `DDS.DomainParticipantFactory.get_instance()`. The purpose of the shutdown hook is to clean up all entities that have been created by the application, but have not been cleaned up during the execution of the application.

It is possible to instruct the API to refrain from installing its shutdown hook by means of the `osplNoDcpsShutdownHook` property. No shutdown hook will be installed if this property is set to "`true`" or "``".

WARNING: If the shutdown hook is disabled, the application is responsible for freeing resources. Not cleaning up resources will lead to memory leaks when the shutdown hook is disabled.

PRISMTECH

2 *DCPS Modules*

DCPS is divided into five modules, which are described briefly in this chapter. Each module consists of several classes as defined at PIM level in the DDS-DCPS specification. Some of the classes as described in the PIM are implemented as a struct in the PSM; these classes are treated as a class in this chapter according to the PIM with a remark about their implementation (struct). In the next chapter their actual implementations are described.

Each class contains several operations, which may be abstract. Those classes, which are implemented as a struct do not have any operations. The modules and the classes are ordered conform the DDS-DCPS specification. The classes, interfaces and operations are described in the next chapter.
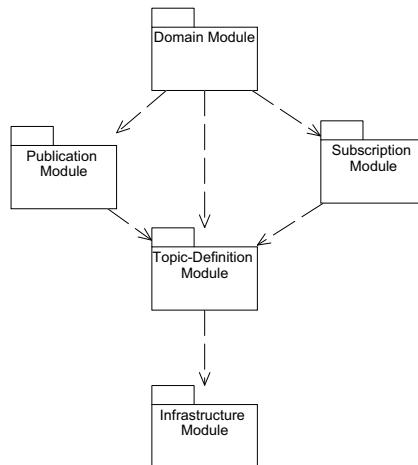
Figure 2 DCPS Module Composition

2.1 **Functionality**

The modules have the following function in the Data Distribution Service:

Infrastructure Module - Defines the abstract classes and interfaces, which are refined by the other modules. It also provides the support for the interaction between the application and the Data Distribution Service (event-based and state-based);

Domain Module - Contains the `DomainParticipant` class, which is the entry point of the application and `DomainParticipantListener` interface;

Topic-Definition Module - Contains the `Topic`, `ContentFilteredTopic` and `MultiTopic` classes. It also contains the `TopicListener` and `ExtTopicListener` interfaces and all support to define `Topic` objects and assign `QosPolicy` settings to them;

Publication Module - Contains the `Publisher` and `DataWriter` classes. It also contains the `PublisherListener` and `DataWriterListener` interfaces;

Subscription Module - Contains the `Subscriber`, `DataReader`, `ReadCondition` and `QueryCondition` classes. It also contains the `SubscriberListener` and `DataReaderListener` interfaces.

2.2 **Infrastructure Module**

This module defines the abstract classes and interfaces, which, in the PIM definition, are refined by the other modules. It also provides the support for the interaction between the application and the Data Distribution Service (event-based and state-based). The event-based interaction is supported by `Listeners`, the state-based interaction is supported by `WaitSets` and `Conditions`.
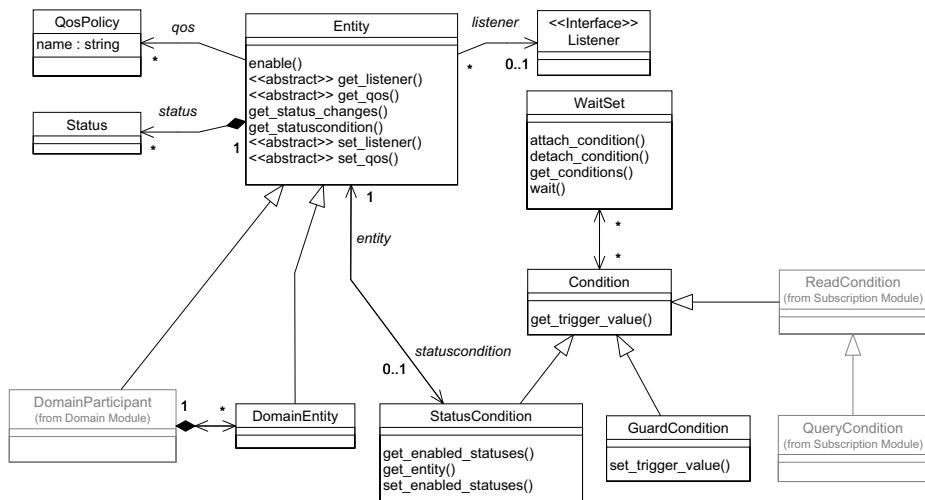
Figure 3 DCPS Infrastructure Module's Class Model

This module contains the following classes:

- `Entity` (abstract)
- `DomainEntity` (abstract)
- `QosPolicy` (abstract, class)

PRISMTECH

- `Listener` (interface)
- `Status` (abstract, class)
- `WaitSet`
- `Condition`
- `GuardCondition`
- `StatusCondition`

2.3 **Domain Module**

This module contains the class `DomainParticipant`, which acts as an entry point of the Data Distribution Service and acts as a factory for many of the classes. The `DomainParticipant` also acts as a container for the other objects that make up the Data Distribution Service. It isolates applications within the same `Domain` from other applications in a different `Domain` on the same set of computers. A `Domain` is a "virtual network" and applications with the same `domainId` are isolated from applications with a different `domainId`. In this way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.
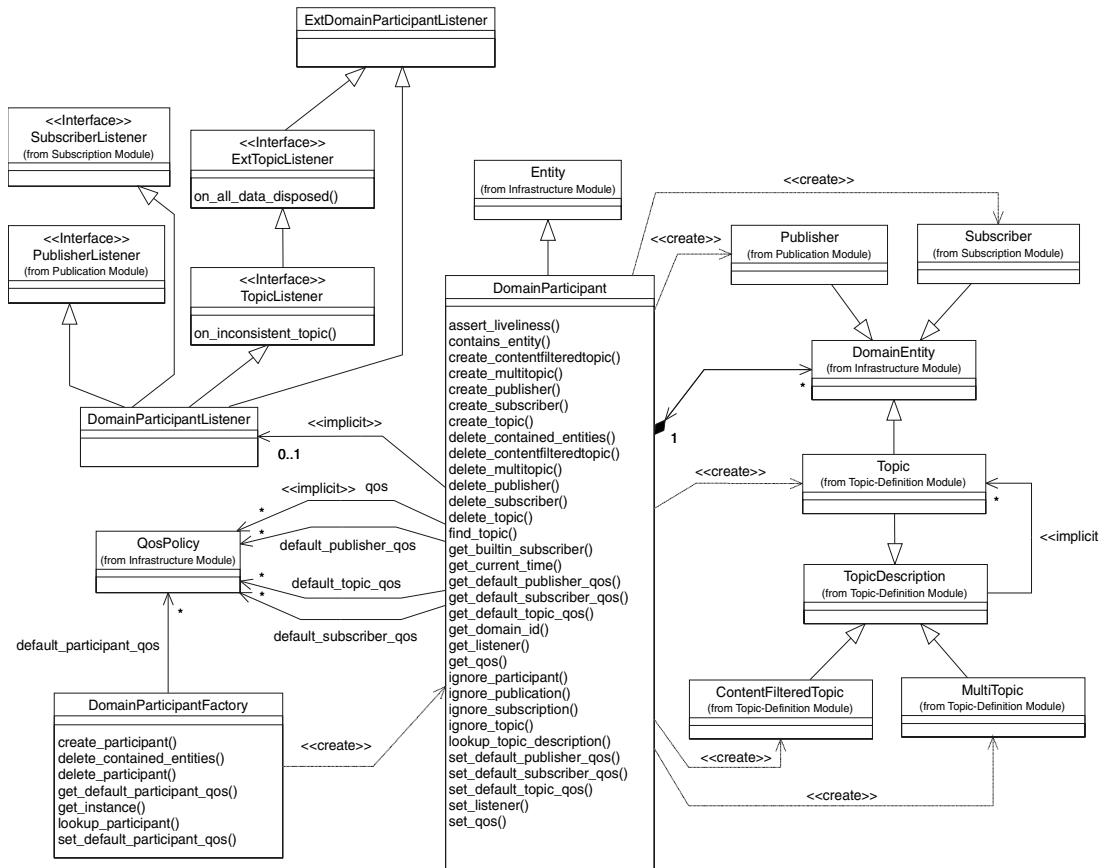
Figure 4 DCPS Domain Module's Class Model

This module contains the following classes:

- `DomainParticipant`
- `DomainParticipantFactory`
- `DomainParticipantListener` (interface)
- `Domain` (*not depicted*)

2.4 Topic-Definition Module

This module contains the `Topic`, `ContentFilteredTopic` and `MultiTopic` classes. It also contains the `TopicListener` interface and all support to define `Topic` objects and assign `QosPolicy` settings to them.

PRISMTECH
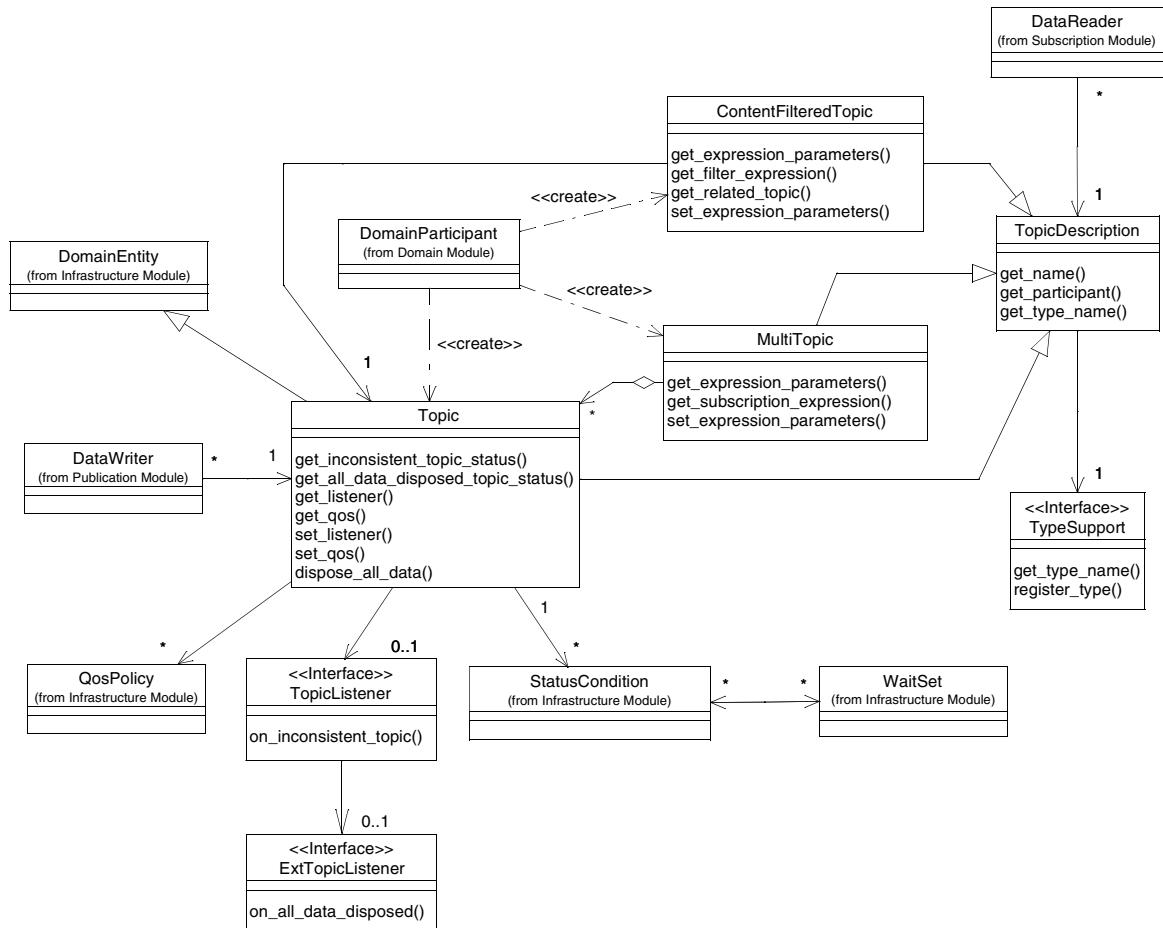
Figure 5 DCPS Topic-Definition Module's Class Model

This module contains the following classes:

- `TopicDescription` (abstract)
- `Topic`
- `ContentFilteredTopic`
- `MultiTopic`
- `TopicListener` (interface)
- ExtTopicListener (interface)
- Topic-Definition type specific classes

"Topic-Definition type specific classes" contains the generic class and the generated data type specific classes. In case of data type `Foo` (this also applies to other types); "Topic-Definition type specific classes" contains the following classes:

- `TypeSupport` (abstract)

- `FooTypeSupport`
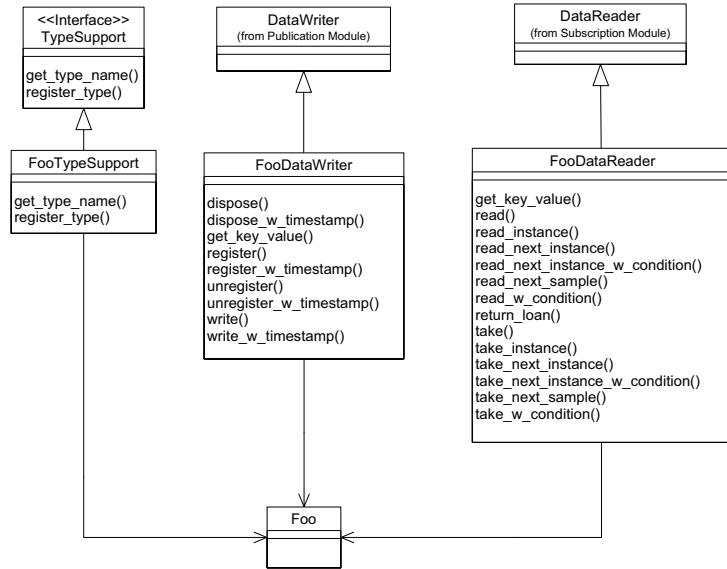
Figure 6 Typed Classes for Data Type "Foo" Pre-processor Generation

2.5 Publication Module

This module supports writing of the data, it contains the `Publisher` and `DataWriter` classes. It also contains the `PublisherListener` and `DataWriterListener` interfaces. Furthermore, it contains all support needed for publication.
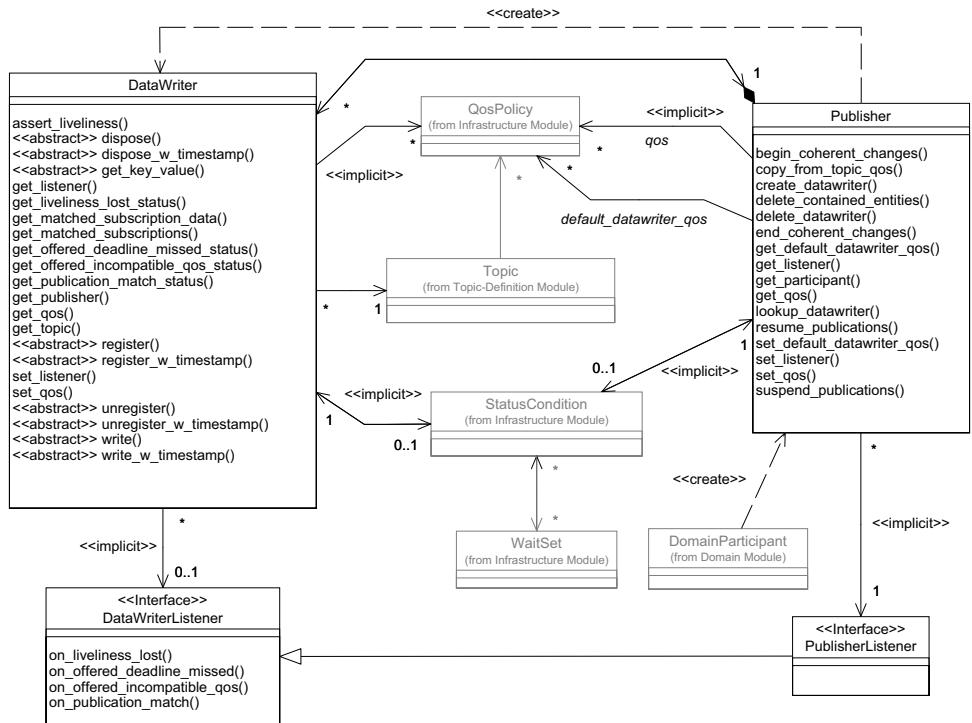
Figure 7 DCPS Publication Module Class Model

This module contains the following classes:

- `Publisher`
- Publication type specific classes
- `PublisherListener` (interface)
- `DataWriterListener` (interface)

"Publication type specific classes" contains the generic class and the generated data type specific classes. In case of data type `Foo` (this also applies to other types); "Publication type specific classes" contains the following classes:

- `DataWriter` (abstract)
- `FooDataWriter`

2.6 Subscription Module

This module supports access to the data, it contains the `Subscriber`, `DataReader`, `ReadCondition` and `QueryCondition` classes. It also contains the `SubscriberListener` and `DataReaderListener` interfaces. Furthermore, it contains all support needed for subscription.
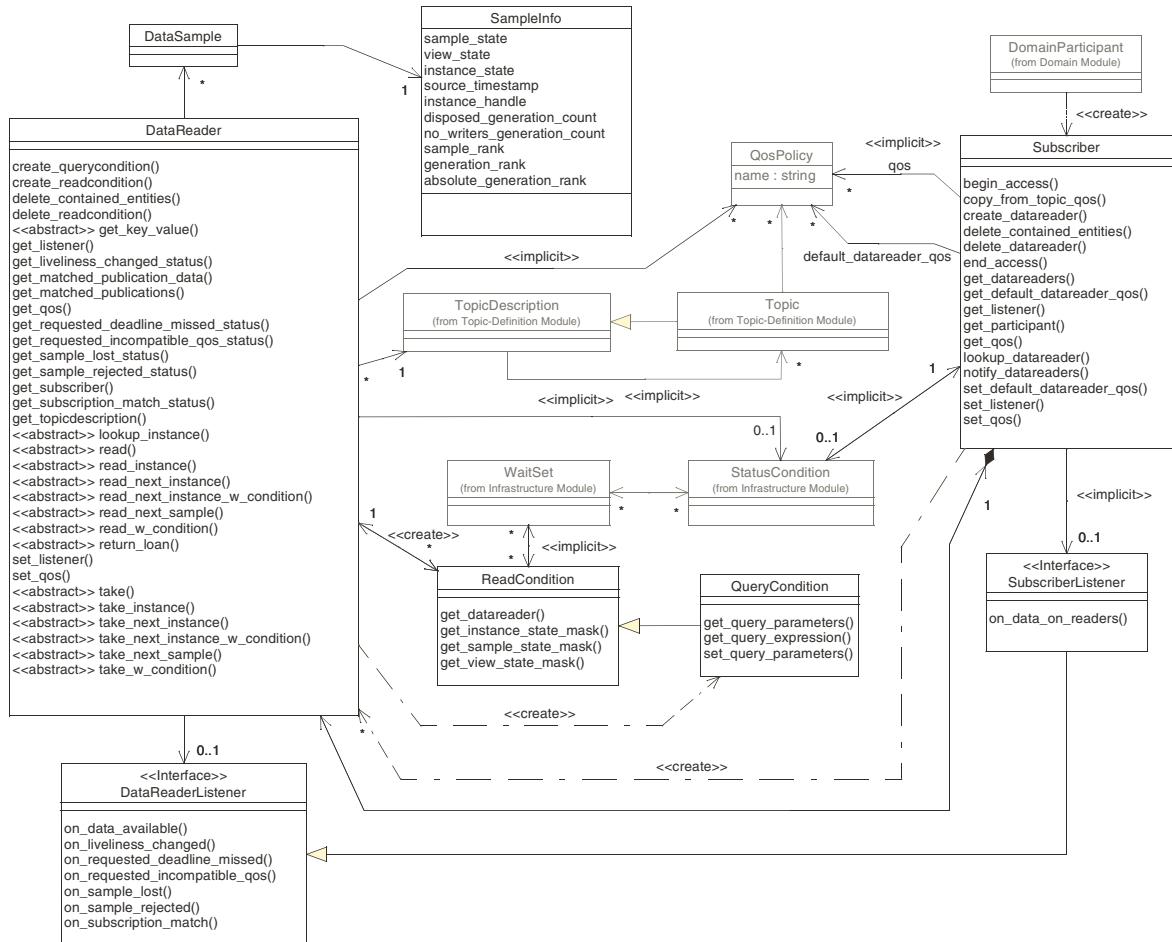
Figure 8 DCPS Subscription Module's Class Model

This module contains the following classes:

- Subscriber
- Subscription type specific classes
- DataSample
- SampleInfo (class)
- SubscriberListener (interface)
- DataReaderListener (interface)
- ReadCondition
- QueryCondition

"Subscription type specific classes" contains the generic class and the generated data type specific classes. In case of data type `Foo` (this also applies to other types); "Subscription type specific classes" contains the following classes:

- `DataReader` (abstract)
- `FooDataReader`

3 *DCPS Classes and Operations*

This chapter describes, for each module, its classes and operations in detail. Each module consists of several classes as defined at PIM level in the DDS-DCPS specification. Some of the classes are implemented as a struct in the PSM. Some of the other classes are abstract, which means they contain some abstract operations.

The Listener *interfaces are designed as an interface at PIM level. In other words, the application must implement the interface operations. Therefore, all* Listener *classes are abstract. A user-defined class for these operations must be provided by the application which must extend from the specific Listener class. All Listener operations must be implemented in the user-defined class. It is up to the application whether an operation is empty or contains some functionality.*

Each class contains several operations, which may be abstract (base class). Abstract operations are not implemented in their base class, but in a type specific class or an application defined class (in case of a Listener*). Classes that are implemented as a struct do not have any operations. Some operations are inherited, which means they are implemented in their base class.*

The abstract operations in a class are listed (including their synopsis), but not implemented in that class. These operations are implemented in their respective derived classes. The interfaces are fully described, since they must be implemented by the application.

3.1 Infrastructure Module
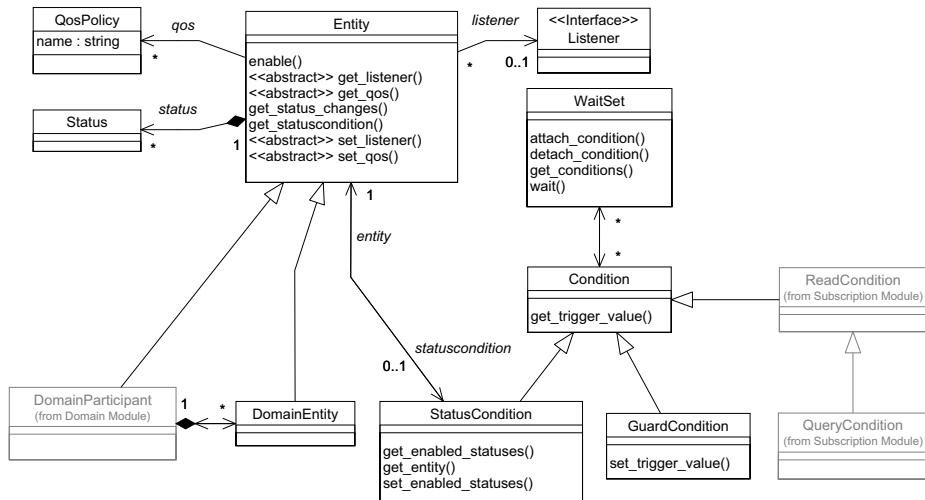
Figure 9 DCPS Infrastructure Module's Class Model

This module contains the following classes:

- `Entity` (interface)
- `DomainEntity` (abstract)
- `QosPolicy` (abstract, class)
- `Listener` (interface)
- `Status` (abstract, class)
- `WaitSet`
- `Condition`
- `GuardCondition`
- `StatusCondition`
- `ErrorInfo`

3.1.1 Interface Entity

Entity is the interface for all the DCPS objects. It acts as a generic interface for `Entity` objects.

The interface description is as follows:

```
public interface Entity
{
//
// abstract operations (implemented in class
```

PRISMTECH

```
                    // DomainParticipant, Topic,
                    // Publisher, DataWriter, Subscriber and DataReader)
                    //
                    // public int
                    //     set_qos
                    //         (EntityQos qos);
                    // public int
                    //     get_qos
                    //         (EntityQosHolder qos);
                    // public int
                    //     set_listener
                    //         (Listener a_listener,
                    //           int mask);
                    // public Listener
                    //     get_listener
                    //         (void);
                    //
                    // implemented API operations
                    //
                       public int
                          enable
                             (void);
                       public StatusCondition
                          get_statuscondition
                             (void);
                       public int
                          get_status_changes
                             (void);
                       public long
                          get_instance_handle
                             (void);
                    };
```

The following paragraphs list all `Entity` operations. The abstract operations are listed but not fully described because they are not implemented in this specific class. The full description of these operations is given in the subclasses, which contain the type specific implementation of these operations.

3.1.1.1 **enable**

Scope

```
DDS.Entity
```

Synopsis

```
import DDS.*;
public int
   enable
      (void);
```

Description

This operation enables the `Entity` on which it is being called when the `Entity` was created with the `EntityFactoryQosPolicy` set to `false`.

Parameters

<none>

Return Value

`int` - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,`
   `RETCODE_OUT_ OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation enables the `Entity`. Created `Entity` objects can start in either an enabled or disabled state. This is controlled by the value of the `EntityFactoryQosPolicy` on the corresponding factory for the `Entity`. Enabled entities are immediately activated at creation time meaning all their immutable QoS settings can no longer be changed. Disabled Entities are not yet activated, so it is still possible to change there immutable QoS settings. However, once activated the immutable QoS settings can no longer be changed.

Creating disabled entities can make sense when the creator of the `Entity` does not yet know which QoS settings to apply, thus allowing another piece of code to set the QoS later on.

The default setting of `EntityFactoryQosPolicy` is such that, by default, entities are created in an enabled state so that it is not necessary to explicitly call `enable` on newly created `entities`.

The `enable` operation is idempotent. Calling `enable` on an already enabled `Entity` returns `RETCODE_OK` and has no effect.

If an `Entity` has not yet been enabled, the only operations that can be invoked on it are: the ones to set, get or copy the `QosPolicy` settings, the ones that set (or get) the listener, the ones that get the `StatusCondition`, the `get_status_changes` operation (although the status of a disabled entity never changes), and the 'factory' operations that create, delete or lookup[1] other `Entities`. Other operations will return the error `RETCODE_NOT_ENABLED`.

`Entities` created from a factory that is disabled, are created disabled regardless of the setting of the `EntityFactoryQosPolicy`.

Calling `enable` on an `Entity` whose factory is not enabled will fail and return `RETCODE_PRECONDITION_NOT_MET`.

---

1.   This includes the lookup_topicdescription, but not the find_topic.

If the `EntityFactoryQosPolicy` has `autoenable_created_entities` set to `true`, the `enable` operation on the factory will automatically enable all `Entities` created from the factory.

The `Listeners` associated with an `Entity` are not called until the `Entity` is enabled. `Conditions` associated with an `Entity` that is not enabled are "inactive", that is, have a `trigger_value` which is `false`.

Return Code

When the operation returns:

- *RETCODE_OK* - the application enabled the `Entity` (or it was already enabled)
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the factory of the `Entity` is not enabled.

3.1.1.2 get_instance_handle

Scope

```
DDS.Entity
```

Synopsis

```
import DDS.*;
public long
    get_instance_handle
        (void);
```

Description

This operation returns the `instance_handle` of the built-in topic sample that represents the specified `Entity`.

Parameters

<none>

Return Value

long - Result value is the `instance_handle` of the built-in topic sample that represents the state of this `Entity`.

Detailed Description

The relevant state of some `Entity` objects are distributed using so-called built-in topics. Each built-in topic sample represents the state of a specific `Entity` and has a unique `instance_handle`. This operation returns the `instance_handle` of the built-in topic sample that represents the specified `Entity`.

Some `Entities` (i.e. `Publisher` and `Subscriber`) do not have a corresponding built-in topic sample, but they still have an `instance_handle` that uniquely identifies the `Entity`.

The `instance_handles` obtained this way can also be used to check whether a specific `Entity` is located in a specific `DomainParticipant`. (See section 3.2.1.2, *contains_entity*, on page 143.)

3.1.1.3 get_listener (abstract)

This abstract operation is defined as a generic operation to access a `Listener`. Each subclass derived from this class, `DomainParticipant`, `Topic`, `Publisher`, `Subscriber`, `DataWriter` and `DataReader` will provide a class specific implementation of this abstract operation.

Synopsis

```
import DDS.*;
public Listener
   get_listener
      (void);
```

3.1.1.4 get_qos (abstract)

This abstract operation is defined as a generic operation to access an object with the `QosPolicy` settings. Each subclass derived from this class, `DomainParticipant`, `Topic`, `Publisher`, `Subscriber`, `DataWriter` and `DataReader` will provide a class specific implementation of this abstract operation.

Synopsis

```
import DDS.*;
public int
   get_qos
      (EntityQosHolder qos);
```

3.1.1.5 get_status_changes

Scope

```
DDS.Entity
```

PRISMTECH

Synopsis

```
import DDS.*;
public int
   get_status_changes
      (void);
```

Description

This operation returns a mask with the communication statuses in the `Entity` that are "triggered".

Parameters

<none>

Return Value

`int` - a bit mask in which each bit shows which value has changed.

Detailed Description

This operation returns a mask with the communication statuses in the `Entity` that are "triggered". That is the set of communication statuses whose value have changed since the last time the application called this operation. This operation shows whether a change has occurred even when the status seems unchanged because the status changed back to the original status.

When the `Entity` is first created or if the `Entity` is not enabled, all communication statuses are in the "un-triggered" state so the mask returned by the operation is empty.

The result value is a bit mask in which each bit shows which value has changed. The relevant bits represent one of the following statuses:

- INCONSISTENT_TOPIC_STATUS
- OFFERED_DEADLINE_MISSED_STATUS
- REQUESTED_DEADLINE_MISSED_STATUS
- OFFERED_INCOMPATIBLE_QOS_STATUS
- REQUESTED_INCOMPATIBLE_QOS_STATUS
- SAMPLE_LOST_STATUS
- SAMPLE_REJECTED_STATUS
- DATA_ON_READERS_STATUS
- DATA_AVAILABLE_STATUS
- LIVELINESS_LOST_STATUS
- LIVELINESS_CHANGED_STATUS
- PUBLICATION_MATCHED_STATUS

- `SUBSCRIPTION_MATCHED_STATUS`
- `ALL_DATA_DISPOSED_TOPIC_STATUS`

Each status bit is declared as a constant and can be used in an `AND` operation to check the status bit against the resulting status mask. Not all statuses are relevant to all `Entity` objects. See the respective `Listener` interfaces for each `Entity` for more information

3.1.1.6 get_statuscondition

Scope

```
DDS.Entity
```

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
      (void);
```

Description

This operation allows access to the `StatusCondition` associated with the `Entity`.

Parameters

<none>

Return Value

StatusCondition - Result value is the `StatusCondition` of the `Entity`.

Detailed Description

Each `Entity` has a `StatusCondition` associated with it. This operation allows access to the `StatusCondition` associated with the `Entity`. The returned condition can then be added to a `WaitSet` so that the application can wait for specific status changes that affect the `Entity`.

3.1.1.7 set_listener (abstract)

This abstract operation is defined as a generic operation to access a `Listener`. Each subclass derived from this class, `DomainParticipant`, `Topic`, `Publisher`, `Subscriber`, `DataWriter` and `DataReader` will provide a class specific implementation of this abstract operation.

Synopsis

```
import DDS.*;
```

```
public int
   set_listener
      (Listener a_listener,
        int mask);
```

3.1.1.8 set_qos (abstract)

This abstract operation is defined as a generic operation to modify an object with the `QosPolicy` settings. Each subclass derived from this class, `DomainParticipant`, `Topic`, `Publisher`, `Subscriber`, `DataWriter` and `DataReader` will provide a class specific implementation of this abstract operation.

Synopsis

```
import DDS.*;
public int
   set_qos
      (EntityQos qos);
```

3.1.2 Class DomainEntity

This class is the abstract base class for the all entities except `DomainParticipant`. The main purpose is to express that `DomainParticipant` is a special kind of `Entity`, which acts as a container of all other `Entity` objects, but cannot contain another `DomainParticipant` within itself. Therefore, this class is not part of the IDL interface in the DCPS PSM description.

The class `DomainEntity` does not contain any operations.
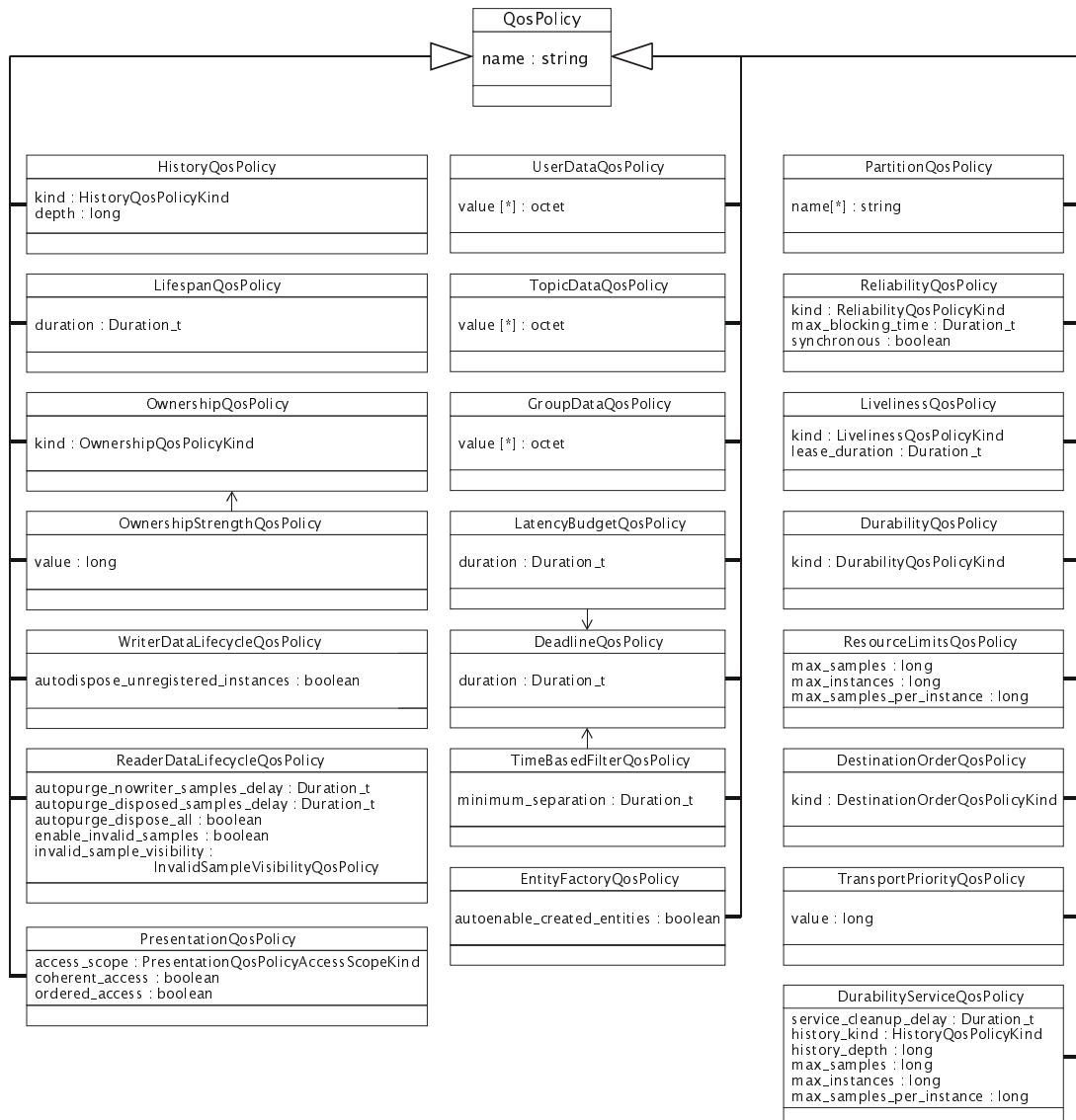
3.1.3 Class QosPolicy

Each `Entity` provides an `<Entity>Qos` class that implements the basic mechanism for an application to specify Quality of Service attributes. This class consists of `Entity` specific `QosPolicy` attributes. `QosPolicy` attributes are class types where each type specifies the information that controls an `Entity` related (configurable) property of the Data Distribution Service.

All `QosPolicies` applicable to an `Entity` are aggregated in a corresponding `<Entity>Qos`, which is a compound class that is set atomically so that it represents a coherent set of `QosPolicy` attributes.

Compound classes are used whenever multiple attributes must be set coherently to define a consistent attribute for a `QosPolicy`.

A full description of any `<Entity>Qos` is given in Appendix A, *Quality Of Service*. The complete list of individual `QosPolicy` settings and their meaning is described below.

Figure 10 QosPolicy Settings

Requested/Offered

In several cases, for communications to occur properly (or efficiently), a `QosPolicy` on the requesting side must be compatible with a corresponding `QosPolicy` on the offering side. For example, if a `DataReader` requests to receive data reliably while the corresponding `DataWriter` defines a best-effort `QosPolicy`, communication will not happen as requested. This means that the

specification for `QosPolicy` follows the Requested/Offered (RxO) pattern while trying to maintain the desirable decoupling of publication and subscription as much as possible. In this pattern:

• the requesting side can specify a "requested" attribute for a particular `QosPolicy`

• the offering side specifies an "offered" attribute for that `QosPolicy`

The Data Distribution Service will then determine whether the attribute requested by the requesting side is compatible with what is offered by the offering side. Only when the two `QosPolicy` settings are compatible, communication is established. If the two `QosPolicy` settings are not compatible, the Data Distribution Service will not establish communication between the two `Entity` objects and notify this fact by means of the `OFFERED_INCOMPATIBLE_QOS` status on the offering side and the `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side. The application can detect this fact by means of a `Listener` or `Condition`.

The interface description of the QosPolicy classes is as follows:

```
// class <Entity>Qos
//
package DDS;
//
//  class <name>QosPolicy
   //
   public final class UserDataQosPolicy
   { public byte value[]; };

   public final class TopicDataQosPolicy
   { public byte value[]; };

   public final class GroupDataQosPolicy
   { public byte value[]; };

   public final class TransportPriorityQosPolicy
   { public int value; };

   public final class LifespanQosPolicy
   { public Duration_t duration; };

   public class DurabilityQosPolicyKind
   { public static final DurabilityQosPolicyKind
         VOLATILE_DURABILITY_QOS;
     public static final DurabilityQosPolicyKind
         TRANSIENT_LOCAL_DURABILITY_QOS;
     public static final DurabilityQosPolicyKind
         TRANSIENT_DURABILITY_QOS;
     public static final DurabilityQosPolicyKind
         PERSISTENT_DURABILITY_QOS; };
```

```
public final class DurabilityQosPolicy
{ public DurabilityQosPolicyKind kind; };

public class PresentationQosPolicyAccessScopeKind
{ public static final PresentationQosPolicyAccessScopeKind
        INSTANCE_PRESENTATION_QOS;
  public static final PresentationQosPolicyAccessScopeKind
        TOPIC_PRESENTATION_QOS;
 public static final PresentationQosPolicyAccessScopeKind
        GROUP_PRESENTATION_QOS; };

public final class PresentationQosPolicy
{ public PresentationQosPolicyAccessScopeKind access_scope;
    public boolean coherent_access;
    public boolean ordered_access; };

public final class DeadlineQosPolicy
{ public Duration_t period; };

public final class LatencyBudgetQosPolicy
{ public Duration_t duration; };

public class OwnershipQosPolicyKind
{ public static final OwnershipQosPolicyKind
        SHARED_OWNERSHIP_QOS;
 public static final OwnershipQosPolicyKind
        EXCLUSIVE_OWNERSHIP_QOS; };

public final class OwnershipQosPolicy
{ public OwnershipQosPolicyKind kind; };

public final class OwnershipStrengthQosPolicy
{ public int value; };

public class LivelinessQosPolicyKind
{ public static final LivelinessQosPolicyKind
        AUTOMATIC_LIVELINESS_QOS;
 public static final LivelinessQosPolicyKind
        MANUAL_BY_PARTICIPANT_LIVELINESS_QOS;
 public static final LivelinessQosPolicyKind
        MANUAL_BY_TOPIC_LIVELINESS_QOS;};

public final class LivelinessQosPolicy
{ public LivelinessQosPolicyKind kind;
    public Duration_t lease_duration; };

public final class TimeBasedFilterQosPolicy
{ public Duration_t minimum_separation; };
```

```
public final class PartitionQosPolicy
{ public String name[]; };

public class ReliabilityQosPolicyKind
{ public static final ReliabilityQosPolicyKind
      BEST_EFFORT_RELIABILITY_QOS;
 public static final ReliabilityQosPolicyKind
      RELIABLE_RELIABILITY_QOS; };

public final class ReliabilityQosPolicy
{ public ReliabilityQosPolicyKind kind;
   public Duration_t max_blocking_time;
   public boolean synchronous; };

public class DestinationOrderQosPolicyKind
{ public static final DestinationOrderQosPolicyKind
      BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS;
 public static final DestinationOrderQosPolicyKind
      BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS;};

public final class DestinationOrderQosPolicy
{ public DestinationOrderQosPolicyKind kind; };

public class HistoryQosPolicyKind
{ public static final HistoryQosPolicyKind KEEP_LAST_HISTORY_QOS;
   public static final
      HistoryQosPolicyKind KEEP_ALL_HISTORY_QOS; };

public final class HistoryQosPolicy
{ public HistoryQosPolicyKind kind;
   public int depth; };

public final class ResourceLimitsQosPolicy
{ public int max_samples;
   public int max_instances;
   public int max_samples_per_instance; };

public final class EntityFactoryQosPolicy
{ public boolean autoenable_created_entities; };

public final class WriterDataLifecycleQosPolicy
{ public boolean autodispose_unregistered_instances; };

public class InvalidSampleVisibilityQosPolicyKind
{ public static final InvalidSampleVisibilityQosPolicyKind
   NO_INVALID_SAMPLES;
  public static final InvalidSampleVisibilityQosPolicyKind
   MINIMUM_INVALID_SAMPLES;
  public static final InvalidSampleVisibilityQosPolicyKind
   ALL_INVALID_SAMPLES; };
```

```
public final class InvalidSampleVisibilityQosPolicy
{ public InvalidSampleVisibilityQosPolicyKind kind; };

public final class ReaderDataLifecycleQosPolicy
{ public Duration_t autopurge_nowriter_samples_delay;
  public Duration_t autopurge_disposed_samples_delay;
  public boolean autopurge_dispose_all;
  public boolean enable_invalid_samples;
  public InvalidSampleVisibilityQosPolicy
    invalid_sample_visibility; };

public final class DurabilityServiceQosPolicy
{ public Duration_t service_cleanup_delay;
   public HistoryQosPolicyKind history_kind;
   public int history_depth;
   public int max_samples;
   public int max_instances;
   public int max_samples_per_instance; };

public class SchedulingClassQosPolicyKind
{ public static final SchedulingClassQosPolicyKind
     SCHEDULE_DEFAULT;
   public static final SchedulingClassQosPolicyKind
     SCHEDULE_TIMESHARING;
   public static final SchedulingClassQosPolicyKind
     SCHEDULE_REALTIME; };

public class SchedulingClassQosPolicy
{ public SchedulingClassQosPolicyKind kind; };

public class SchedulingPriorityQosPolicyKind
{ public static final SchedulingPriorityQosPolicyKind
     PRIORITY_RELATIVE;
   public static final SchedulingPriorityQosPolicyKind
     PRIORITY_ABSOLUTE; };

public class SchedulingPriorityQosPolicy
{ public SchedulingPriorityQosPolicyKind kind; };

public class SchedulingQosPolicy
{ public SchedulingClassQosPolicy scheduling_class;
   public SchedulingPriorityQosPolicy scheduling_priority_kind;
   public int scheduling_priority; };
public class SubscriptionKeyQosPolicy
{  public boolean use_key_list;
   public String[] key_list; };

public class ReaderLifespanQosPolicy
{  public boolean use_lifespan;
```

PRISMTECH

```
                    public Duration_t duration; };

            public class ShareQosPolicy
            {   public boolean enable;
                public String name; };

            public class ViewKeyQosPolicy
            {   public boolean use_key_list;
                public String[] key_list;  };
```

Default Attributes

The default value of each `QosPolicy` attribute are listed in the next table:

Table 3 QosPolicy Default Attributes

QosPolicy	Attribute	Value
user_data	value.length	0
topic_data	value.length	0
group_data	value.length	0
transport_priority	value	0
lifespan	duration	DURATION_INFINITE
durability	kind	VOLATILE_DURABILITY_QOS
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	false
	ordered_access	false
deadline	period	DURATION_INFINITE
latency_budget	duration	0
ownership_strength	value	0
ownership	kind	SHARED_OWNERSHIP_QOS
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITE
time_based_filter	minimum_separation	0
partition	name.length	0
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	false
destination_order	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS

Table 3 QosPolicy Default Attributes (continued)

QosPolicy	Attribute	Value
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
entity_factory	autoenable_created_entities	true
writer_data_lifecycle	autodispose_ unregistered_instances	true
reader_data_lifecycle	autopurge_ nowriter_samples_delay	DURATION_INFINITE
	autopurge_ disposed_samples_delay	DURATION_INFINITE
	autopurge_dispose_all	FALSE
	enable_invalid_samples	true
	invalid_sample_visibility .kind	MINIMUM_INVALID_SAMPLES
durability_service	history_kind	KEEP_LAST
	history_depth	1
	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
	service_cleanup_delay	0
watchdog_scheduling, listener_scheduling	scheduling_class.kind	SCHEDULE_DEFAULT
	scheduling_priority_kind. kind	PRIORITY_RELATIVE
	scheduling_priority	0
subscription_keys	use_key_list	false
	key_list.length	0
reader_lifespan	use_lifespan	false
	duration	DURATION_INFINITE
share	name	""
	enable	false
view_keys	use_key_list	false
	key_list.length	0

PRISMTECH

RxO

The `QosPolicy` settings that need to be set in a compatible manner between the publisher and subscriber ends are indicated by the setting of the "RxO" (Requested/Offered) property. The "RxO" property of each `QosPolicy` is listed in Table 4, *QosPolicy Basics*, on page 44:

- A "RxO" setting of "Yes" indicates that the `QosPolicy` can be set at both ends (publishing and subscribing) and the attributes must be set in a compatible manner. In this case the compatible attributes are explicitly defined

- A "RxO" setting of "No" indicates that the `QosPolicy` can be set at both ends (publishing and subscribing) but the two settings are independent. That is, all combinations of attributes are compatible

- A "RxO" setting of "Not applicable" indicates that the `QosPolicy` can only be specified at either the publishing or the subscribing end, but not at both ends. So compatibility does not apply

Changeable

The "changeable" property determines whether the `QosPolicy` can be changed after the `Entity` is enabled. In other words, a `QosPolicy` with "changeable" setting of "No" is considered "immutable" and can only be specified either at `Entity` creation time or prior to calling the enable operation on the `Entity`.

When the application tries to change a `QosPolicy` with "changeable" setting of "No", the Data Distribution Service will notify this by returning a `RETCODE_IMMUTABLE_POLICY`.

The basic way to modify or set the `<Entity>Qos` is by using a `get_qos` and `set_qos` operation to get all `QosPolicy` settings from this `Entity` (that is the `<Entity>Qos`), modify several specific `QosPolicy` settings and put them back using an user operation to set all `QosPolicy` settings on this `Entity` (that is the `<Entity>Qos`). An example of these operations for the `DataWriter` are `get_qos` and `set_qos`, which take the `<Entity>Qos` as a parameter.

The "RxO" setting and the "changeable" setting of each `QosPolicy` are listed in the next table:

Table 4 QosPolicy Basics

QosPolicy	Concerns Entity	RxO	Changeable After Enabling
user_data	DomainParticipant DataReader DataWriter	No	Yes
topic_data	Topic	No	Yes
group_data	Publisher Subscriber	No	Yes
transport_priority	Topic DataWriter	Not applicable	Yes
lifespan	Topic DataWriter	Not applicable	Yes
durability	Topic DataReader DataWriter	Yes	No
presentation	Publisher Subscriber	Yes	No
deadline	Topic DataReader DataWriter	Yes	Yes
latency_budget	Topic DataReader DataWriter	Yes	Yes
ownership	Topic DataReader DataWriter	Yes	No
ownership_strength	DataWriter	Not applicable	Yes
liveliness	Topic DataReader DataWriter	Yes	No
time_based_filter	DataReader	Not applicable	Yes
partition	Publisher Subscriber	No	Yes
reliability	Topic DataReader DataWriter	Yes	No

PRISMTECH

Table 4 QosPolicy Basics (continued)

QosPolicy	Concerns Entity	RxO	Changeable After Enabling
destination_order	Topic DataReader DataWriter	Yes	No
history	Topic DataReader DataWriter	No	No
resource_limits	Topic DataReader DataWriter	No	No
entity_factory	DomainParticipantFactory DomainParticipant Publisher Subscriber	No	Yes
writer_data_lifecycle	DataWriter	Not applicable	Yes
reader_data_lifecycle	DataReader	Not applicable	Yes
durability_service	Topic	No	No
scheduling	DomainParticipant	Not applicable	No
subscription_keys	DataReader	Not applicable	No
reader_lifespan	DataReader	Not applicable	Yes
share	DataReader Subscriber	Not applicable	No No
view_keys	DataReaderView	Not applicable	No

The following paragraphs describe each `<name>QosPolicy` class.

3.1.3.1 DeadlineQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class DeadlineQosPolicy
{ public Duration_t period; }
```

Description

This `QosPolicy` defines the period within which a new sample is expected by the `DataReader` or to be written by the `DataWriter`.

Attributes

Duration_t period - specifies the period within which a new sample is expected or to be written.

Detailed Description

This `QosPolicy` will set the period within which a `DataReader` expects a new sample or, in case of a `DataWriter`, the period in which it expects applications to write the sample. The default value of the `period` is `DURATION_INFINITE`, indicating that there is no deadline. The `QosPolicy` may be used to monitor the real-time behaviour, a `Listener` or a `StatusCondition` may be used to catch the event that is generated when a deadline is missed.

`DeadlineQosPolicy` is instance oriented (i.e. the period is monitored for each individual instance).

The exact consequences of a missed deadline depend on the Entity in which it occured, and the `OwnershipQosPolicy` value of that Entity:

- In case a `DataWriter` misses an instance deadline (regardless of its `OwnershipQosPolicy` setting), an `offered_deadline_missed_status` is raised, which can be detected by either a `Listener` or a `StatusCondition`. There are no further consequences.

- In case a `DataReader` misses an instance deadline, a `requested_deadline_missed_status` is raised, which can be detected by either a `Listener` or a `StatusCondition`. In case the `OwnershipQosPolicy` is set to `SHARED`, there are no further consequences. In case the `OwnershipQosPolicy` is set to `EXCLUSIVE`, the ownership of that instance on that particular `DataReader` is transferred to the next available highest strength `DataWriter`, but this will have no impact on the `instance_state` whatsoever. So even when a deadline is missed for an instance that has no other (lower-strength) DataWriters to transfer ownership to, the `instance_state` remains unchanged. See also Section 3.1.3.11, *OwnershipQosPolicy*.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` may be changed by using the `set_qos` operation.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 5 DeadlineQosPolicy

Period	Compatibility
`offered period < requested period`	compatible
`offered period = requested period`	compatible
`offered period > requested period`	INcompatible

Whether communication is established, is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. In other words, the communication between any `DataWriter` and `DataReader` depends on what is expected by the `DataReader`. As a consequence, a `DataWriter` that has an incompatible `QoS` with respect to what a `DataReader` specifies, is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible `QoS` with respect to what a `DataWriter` specifies, does not get any data from that particular `DataWriter`.

Changing an existing deadline period using the `set_qos` operation on either the `DataWriter` or `DataReader` may have consequences for the connectivity between readers and writers, depending on their `RxO` values. (See also in Section 3.1.3, *Class QosPolicy*, the paragraph entitled *Requested/Offered*.) Consider a writer with deadline period `Pw` and a reader with deadline period `Pr`, where `Pw <= Pr`. In this case a connection between that reader and that writer is established. Now suppose `Pw` is changed so that `Pw > Pr`, then the existing connection between reader and writer will be lost, and the reader will behave as if the writer unregistered all its instances, transferring the ownership of these instances when appropriate. See also Section 3.1.3.11, *OwnershipQosPolicy*.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way, the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.2 **DestinationOrderQosPolicy**

Scope

DDS

Synopsis

```
import DDS.*
   public class DestinationOrderQosPolicyKind
   { public static final DestinationOrderQosPolicyKind
         BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS;
      public static final DestinationOrderQosPolicyKind
                  BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS;
   }
   public class DestinationOrderQosPolicy
   { public DestinationOrderQosPolicyKind kind; }
```

Description

This `QosPolicy` controls the order in which the `DataReader` stores the data.

Attributes

DestinationOrderQosPolicyKind kind - controls the order in which the `DataReader` stores the data.

Detailed Description

This `QosPolicy` controls the order in which the `DataReader` stores the data. The order of storage is controlled by the timestamp. However a choice can be made to use the timestamp of the `DataReader` (time of reception) or the timestamp of the `DataWriter` (source timestamp).

This `QosPolicy` is applicable to a `DataWriter`, `DataReader` and a `Topic`. After enabling of the concerning entity, this `QosPolicy` cannot be changed any more.

Attribute

The `QosPolicy` is controlled by the attribute `kind` which may be:

- *BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS*
- *BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS*

When set to `BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS`, the order is based on the timestamp, at the moment the sample was received by the `DataReader`.

When set to `BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS`, the order is based on the timestamp, which was set by the `DataWriter`. This means that the system needs some time synchronisation.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

PRISMTECH

Table 6 Requested/Offered DestinationOrderQosPolicy

Requested Offered	BY_RECEPTION_TIMESTAMP	BY_SOURCE_TIMESTAMP
BY_RECEPTION_TIMESTAMP	compatible	INcompatible
BY_SOURCE_TIMESTAMP	compatible	compatible

Whether communication is established, it is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. In other words, the communication between any `DataWriter` and `DataReader` depends on what is expected by the `DataReader`. As a consequence, a `DataWriter` that has an incompatible QoS with respect to what a `DataReader` specified, is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible QoS with respect to what a `DataWriter` specified, does not get any data from that particular `DataWriter`.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way, the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.3 **DurabilityQosPolicy**

Scope

    DDS

Synopsis

```
import DDS.*;
public class DurabilityQosPolicyKind
{
 public static final DurabilityQosPolicyKind
               VOLATILE_DURABILITY_QOS;
 public static final DurabilityQosPolicyKind
               TRANSIENT_LOCAL_DURABILITY_QOS;
 public static final DurabilityQosPolicyKind
               TRANSIENT_DURABILITY_QOS;
 public static final DurabilityQosPolicyKind
               PERSISTENT_DURABILITY_QOS;
}
public class DurabilityQosPolicy
{
 public DurabilityQosPolicyKind kind;
}
```

Description

This `QosPolicy` controls whether the data should be stored for late joining readers.

Attributes

`DurabilityQosPolicyKind kind` - specifies the type of durability from
VOLATILE_DURABILITY_QOS (short life) to PERSISTENT_DURABILITY_QOS
(long life).

Detailed Description

The decoupling between `DataReader` and `DataWriter` offered by the Data
Distribution Service allows an application to write data even if there are no current
readers on the network. Moreover, a `DataReader` 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 some history. This `QosPolicy` controls whether the
Data Distribution Service will actually make data available to late-joining
`DataReaders`.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After
enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Attributes

The `QosPolicy` is controlled by the attribute `kind` which may be:

- *VOLATILE_DURABILITY_QOS* - the samples are not available to late-joining
  `DataReaders`. In other words, only `DataReaders`, which were present at the
  time of the writing and have subscribed to this `Topic`, will receive the sample.
  When a `DataReader` subscribes afterwards (late-joining), it will only be able to
  read the next written sample. This setting is typically used for data, which is
  updated quickly

- *TRANSIENT_LOCAL_DURABILITY_QOS* - currently behaves identically to the
  TRANSIENT_DURABILITY_QOS, except for its RxO properties. The desired
  behaviour of TRANSIENT_LOCAL_DURABILITY_QOS can be achieved from the
  TRANSIENT_DURABILITY_QOS with the default (TRUE) setting of the
  `autodispose_unregistered_instances` flag on the `DataWriter` and the
  `service_cleanup_delay` set to 0 on the durability service. This is because for
  TRANSIENT_LOCAL, the data should only remain available for late-joining
  readers during the lifetime of its source writer, so it is not required to survive after
  its source writer has been deleted. Since the deletion of a writer implicitly
  unregisters all its instances, an `autodispose_unregistered_instances`
  value of TRUE will also dispose the affected data from the durability store, and
  thus prevent it from remaining available to late joining readers.

PRISMTECH

- *TRANSIENT_DURABILITY_QOS* - some samples are available to late-joining `DataReaders` (stored in memory). This means that the late-joining `DataReaders` are able to read these previously written samples. The `DataReader` does not necessarily have to exist at the time of writing. Not all samples are stored (depending on `QosPolicy History` and `QosPolicy resource_limits`). The storage does not depend on the `DataWriter` and will outlive the `DataWriter`. This may be used to implement reallocation of applications because the data is saved in the Data Distribution Service (not in the `DataWriter`). This setting is typically used for state related information of an application. In this case also the `DurabilityServiceQosPolicy` settings are relevant for the behaviour of the Data Distribution Service

- *PERSISTENT_DURABILITY_QOS* - the data is stored in permanent storage (e.g. hard disk). This means that the samples are also available after a system restart. The samples not only outlives the `DataWriters`, but even the Data Distribution Service and the system. This setting is typically used for attributes and settings for an application or the system. In this case also the `DurabilityServiceQosPolicy` settings are relevant for the behaviour of the Data Distribution Service.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 7 Requested/Offered DurabilityQosPolicy

Requested Offered	**VOLATILE**	**TRANSIENT_ LOCAL**	**TRANSIENT**	**PERSISTENT**
VOLATILE	compatible	INcompatible	INcompatible	INcompatible
TRANSIENT_LOCAL	compatible	compatible	INcompatible	INcompatible
TRANSIENT	compatible	compatible	compatible	INcompatible
PERSISTENT	compatible	compatible	compatible	compatible

This means that the Request/Offering mechanism is applicable between:

- the `DataWriter` and the `DataReader`. If the `QosPolicy` settings between `DataWriter` and `DataReader` are inconsistent, no communication between them is established. In addition the `DataWriter` will be informed via a `REQUESTED_INCOMPATIBLE_QOS` status change and the `DataReader` will be informed via an `OFFERED_INCOMPATIBLE_QOS` status change;

- the `DataWriter` and the Data Distribution Service (as a built-in `DataReader`). If the `QosPolicy` settings between `DataWriter` and the Data Distribution Service are inconsistent, no communication between them is established. In that case data published by the `DataWriter` will not be maintained by the service and as a consequence will not be available for late joining `DataReaders`. The `QosPolicy` of the Data Distribution Service in the role of `DataReader` is specified by the `Topic QosPolicy`

- the Data Distribution Service (as a built-in `DataWriter`) and the `DataReader`. If the `QosPolicy` settings between the Data Distribution Service and the `DataReader` are inconsistent, no communication between them is established. In that case the Data Distribution Service will not publish historical data to late joining `DataReaders`. The `QosPolicy` of the Data Distribution Service in the role of `DataWriter` is specified by the `Topic QosPolicy`

Cleanup

The `DurabilityQosPolicy` kind setting `TRANSIENT_LOCAL_DURABILITY_QOS`, `TRANSIENT_DURABILITY_QOS` and `PERSISTENT_DURABILITY_QOS` determine that the `DurabilityServiceQosPolicy` applies for the `Topic`. It controls amongst others at which time the durability service is allowed to remove all information regarding a data-instance. Information on a data-instance is maintained until the following conditions are met:

- the instance has been explicitly disposed of (`instance_state = NOT_ALIVE_DISPOSED_INSTANCE_STATE`)

- *and* the system detects that there are no more "live" `DataWriter` objects writing the instance, that is, all `DataWriter` either `unregister_instance` the instance (call `unregister_instance` operation) or lose their `liveliness`

- *and* a time interval longer than `service_cleanup_delay` has elapsed since the moment the Data Distribution Service detected that the previous two conditions were met

The use of the `DurabilityServiceQosPolicy` attribute `service_cleanup_delay` is apparent in the situation where an application disposes of an `instance` and it crashes before having a chance to complete additional tasks related to the disposition. Upon re-start the application may ask for initial data to regain its state and the delay introduced by the `service_cleanup_delay` allows the re-started application to receive the information on the disposed of `instance` and complete the interrupted tasks.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.4 DurabilityServiceQosPolicy

Scope

    DDS

Synopsis

```
import DDS.*;
public final class DurabilityServiceQosPolicy
{
 public Duration_t service_cleanup_delay;
 public HistoryQosPolicyKind history_kind;
 public int history_depth;
 public int max_samples;
 public int max_instances;
 public int max_samples_per_instance; };
```

Description

This `QosPolicy` controls the behaviour of the durability service regarding transient and persistent data.

Attributes

Duration_t service_cleanup_delay - specifies how long the durability service must wait before it is allowed to remove the information on the transient or persistent topic data-instances as a result of incoming dispose messages.

HistoryQosPolicyKind history_kind - specifies the type of history, which may be `KEEP_LAST_HISTORY_QOS` or `KEEP_ALL_HISTORY_QOS`, the durability service must apply for the transient or persistent topic data-instances.

int history_depth - specifies the number of samples of each instance of data (identified by its key) that is managed by the durability service for the transient or persistent topic data-instances. If `history_kind` is `KEEP_LAST_HISTORY_QOS`, `history_depth` must be smaller than or equal to `max_samples_per_instance` for this `QosPolicy` to be consistent.

int max_samples - specifies the maximum number of data samples for all instances the durability service will manage for the transient or persistent topic data-instances.

int max_instances - specifies the maximum number of instances the durability service - manage for the transient or persistent topic data-instances.

int max_samples_per_instance - specifies the maximum number of samples of any single instance the durability service will manage for the transient or persistent topic data-instances. If history_kind is KEEP_LAST_HISTORY_QOS, max_samples_per_instance must be greater than or equal to history_depth for this QosPolicy to be consistent.

Detailed Description

This QosPolicy controls the behaviour of the durability service regarding transient and persistent data. It controls for the transient or persistent topic; the time at which information regarding the topic may be discarded, the history policy it must set and the resource limits it must apply.

Cleanup

The setting of the DurabilityServiceQosPolicy only applies when kind of the DurabilityQosPolicy is either TRANSIENT_DURABILITY_QOS or PERSISTENT_DURABILITY_QOS. The service_cleanup_delay setting controls at which time the durability service" is allowed to remove all information regarding a data-instance. Information on a data-instance is maintained until the following conditions are met:

- the instance has been explicitly disposed of (instance_state = NOT_ALIVE_DISPOSED_INSTANCE_STATE)

- *and* the system detects that there are no more "live" DataWriter objects writing the instance, that is, all DataWriter either unregister_instance the instance (call unregister_instance operation) or lose their liveliness

- *and* a time interval longer than service_cleanup_delay has elapsed since the moment the Data Distribution Service detected that the previous two conditions were met.

The use of the attribute service_cleanup_delay is apparent in the situation where an application disposes of an instance and it crashes before having a chance to complete additional tasks related to the disposition. Upon re-start the application may ask for initial data to regain its state and the delay introduced by the service_cleanup_delay allows the re-started application to receive the information on the disposed of instance and complete the interrupted tasks

PRISMTECH

History

The attributes `history_kind` and `history_depth` apply to the history settings of the Durability Service's internal `DataWriter` and `DataReader` managing the topic. The `HistoryQosPolicy` behaviour, as described in Section 3.1.3.7, *HistoryQosPolicy*, on page 57 applies to these attributes.

Resource Limits

The attributes `max_samples`, `max_instances` and `max_samples_ per_instance` apply to the resource limits of the Durability Service's internal `DataWriter` and `DataReader` managing the topic. The `ResourceLimitsQosPolicy` behaviour, as described in paragraph 3.1.3.17 (ResourceLimitsQosPolicy) applies to these attributes.

TopicQos

This `QosPolicy` can be set on a `Topic` only. After enabling of the concerning `Topic`, this `QosPolicy` can not be changed any more.

3.1.3.5 EntityFactoryQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class EntityFactoryQosPolicy
        { public boolean autoenable_created_entities; };
```

Description

This `QosPolicy` controls the behaviour of the `Entity` as a factory for other entities.

Attributes

boolean autoenable_created_entities - specifies whether the entity acting as a factory automatically enables the instances it creates. If autoenable_created_entities is true the factory will automatically enable each created `Entity`, otherwise it will not.

Detailed Description

This `QosPolicy` controls the behaviour of the `Entity` as a factory for other entities. It concerns only `DomainParticipantFactory` (as factory for `DomainParticipant`), `DomainParticipant` (as a factory for `Publisher`, `Subscriber`, and `Topic`), `Publisher` (as factory for `DataWriter`), and `Subscriber` (as factory for `DataReader`).

This policy is mutable. A change in the policy affects only the entities created after the change; not the previously created entities.

The setting of `autoenable_created_entities` to `true` indicates that the factory `create_<entity>` operation will automatically invoke the `enable` operation each time a new `Entity` is created. Therefore, the `Entity` returned by `create_<entity>` will already be enabled. A setting of `false` indicates that the `Entity` will not be automatically enabled: the application will need to enable it explicitly by means of the `enable` operation. See paragraph 3.1.1.1 (enable) for a detailed description about the differences between enabled and disabled entities.

The default setting of `autoenable_created_entities` is `true` meaning that by default it is not necessary to explicitly call `enable` on newly created entities.

3.1.3.6 GroupDataQosPolicy

Scope

    DDS

Synopsis

```
import DDS.*;
        public final class GroupDataQosPolicy
        { public byte value[]; }
```

Description

This `QosPolicy` allows the application to attach additional information to a Publisher or Subscriber Entity. This information is distributed with the BuiltinTopics.

Attributes

byte value[] - a sequence of bytes that holds the application group data. By default, the sequence has length 0.

Detailed Description

This `QosPolicy` allows the application to attach additional information to a Publisher or Subscriber Entity. This information is distributed with the `BuiltinTopic`. An application that discovers a new Entity of the listed kind, can

PRISMTECH

use this information to add additional functionality. The `GroupDataQosPolicy` is changeable and updates of the BuiltinTopic instance must be expected. Note that the Data Distribution Service is not aware of the real structure of the group data (the Data Distribution System handles it as an opaque type) and that the application is responsible for correct mapping on structural types for the specific platform.

3.1.3.7 **HistoryQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class HistoryQosPolicyKind
{
 public static final HistoryQosPolicyKind
    KEEP_LAST_HISTORY_QOS;
 public static final HistoryQosPolicyKind
     KEEP_ALL_HISTORY_QOS;
}
public final class HistoryQosPolicy
{
 public HistoryQosPolicyKind kind;
 public int depth;
}
```

Description

This `QosPolicy` controls which samples will be stored when the value of an instance changes (one or more times) before it is finally communicated.

Attributes

HistoryQosPolicyKind kind - specifies the type of history, which may be KEEP_LAST_HISTORY_QOS or KEEP_ALL_HISTORY_QOS.

int depth - specifies the number of samples of each instance of data (identified by its key) managed by this `Entity`.

Detailed Description

This `QosPolicy` controls whether the Data Distribution Service should deliver only the most recent sample, attempt to deliver all samples, or do something in between. In other words, how the `DataWriter` or `DataReader` should store samples. Normally, only the most recent sample is available but some history can be stored.

DataWriter

On the publishing side this `QosPolicy` controls the samples that should be maintained by the `DataWriter` on behalf of existing `DataReader` objects. The behaviour with respect to a `DataReader` objects discovered after a sample is written is controlled by the `DurabilityQosPolicy`.

DataReader

On the subscribing side it controls the samples that should be maintained until the application "takes" them from the Data Distribution Service.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Attributes

The `QosPolicy` is controlled by the attribute `kind` which can be:

- `KEEP_LAST_HISTORY_QOS` - the Data Distribution Service will only attempt to keep the latest values of the instance and discard the older ones. The attribute "`depth`" determines how many samples in history will be stored. In other words, only the most recent samples in history are stored. On the publishing side, the Data Distribution Service will only keep the most recent "`depth`" samples of each instance of data (identified by its `key`) managed by the `DataWriter`. On the subscribing side, the `DataReader` will only keep the most recent "`depth`" samples received for each instance (identified by its `key`) until the application "takes" them via the `DataReader.take` operation.

- `KEEP_LAST_HISTORY_QOS` - is the default `kind`. The default value of `depth` is 1, indicating that only the most recent value should be delivered. If a `depth` other than 1 is specified, it should be compatible with the settings of the `ResourcelimitsQosPolicy max_samples_per_instance`. For these two `QosPolicy` settings to be compatible, they must verify that `depth   <= max_samples_per_instance`, otherwise a `RETCODE_INCONSISTENT_ POLICY` is generated on relevant operations

- `KEEP_ALL_HISTORY_QOS` - all samples are stored, provided, the resources are available. On the publishing side, the Data Distribution Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its `key`) managed by the `DataWriter` until they can be delivered to all `subscribers`.

  On the subscribing side, the Data Distribution Service will attempt to keep all samples of each instance of data (identified by its `key`) managed by the `DataReader`. These samples are kept until the application "takes" them from the Data Distribution Service via the `DataReader.take` operation. The setting of `depth` has no effect. Its implied value is `LENGTH_UNLIMITED`. The resources that the Data Distribution Service can use to keep this history are limited by the

settings of the `ResourceLimitsQosPolicy`. If the limit is reached, the behaviour of the Data Distribution Service will depend on the `ReliabilityQosPolicy`.

If the `ReliabilityQosPolicy` is `BEST_EFFORT_RELIABILITY_QOS`, the old values are discarded. If `ReliabilityQosPolicy` is `RELIABLE_RELIABILITY_QOS`, the Data Distribution Service will block the `DataWriter` until it can deliver the necessary old values to all `subscribers`

On the subscribing side it controls the samples that should be maintained until the application "takes" them from the Data Distribution Service. On the publishing side this `QosPolicy` controls the samples that should be maintained by the `DataWriter` on behalf of `DataReader` objects. The behaviour with respect to a `DataReader` objects discovered after a sample is written is controlled by the `DurabilityQosPolicy`. In more detail, this `QosPolicy` specifies the behaviour of the Data Distribution Service in case the value of a sample changes (one or more times) before it can be successfully communicated to one or more `Subscribers`.

Requested/Offered

The setting of the `QosPolicy` offered is independent of the one requested, in other words they are never considered incompatible. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side or `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side will not be raised.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.8 **LatencyBudgetQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class LatencyBudgetQosPolicy
{
 public Duration_t duration;
}
```

Description

Specifies the maximum acceptable additional delay to the typical transport delay from the time the data is written until the data is delivered at the DataReader and the application is notified of this fact.

Attributes

Duration_t duration - specifies the maximum acceptable additional delay from the time the data is written until the data is delivered.

Detailed Description

This `QosPolicy` specifies the maximum acceptable additional delay to the typical transport delay from the time the data is written until the data is delivered at the `DataReader` and the application is notified of this fact. This `QosPolicy` provides a means for the application to indicate to the Data Distribution Service the "urgency" of the data-communication. By having a non-zero duration the Data Distribution Service can optimize its internal operation. The default value of the `duration` is zero, indicating that the delay should be minimized.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` may be changed by using the `set_qos` operation.

Requested/Offered

This `QosPolicy` is considered a hint to the Data Distribution Service, which will automatically adapt its behaviour to meet the requirements of the shortest delay if possible. In case the Requested/Offered QosPolicy are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 8 LatencyBudgetQosPolicy

Duration	Compatibility
offered duration < requested duration	compatible
offered duration = requested duration	compatible
offered duration > requested duration	INcompatible

Note that even when the offered duration is considered compatible to the requested duration, this duration is not enforced in any way: there will be no notification on any violations of the requested duration.

PRISMTECH

Changing an existing latency budget using the `set_qos` operation on either the `DataWriter` or `DataReader` may have consequences for the connectivity between readers and writers, depending on their RxO values. (See also in Section 3.1.3, *Class QosPolicy*, the paragraph entitled *Requested/Offered*.) Consider a writer with budget Bw and a reader with budget Br, where Bw `<=` Br. In this case a connection between that reader and that writer is established. Now suppose Bw is changed so that Bw `>` Br, then the existing connection between reader and writer will be lost, and the reader will behave as if the writer unregistered all its instances, transferring the ownership of these instances when appropriate. See also Section 3.1.3.11, *OwnershipQosPolicy*.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.9 LifespanQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class LifespanQosPolicy
{
 public Duration_t duration;
}
```

Description

This `QosPolicy` specifies the duration of the validity of the data written by the `DataWriter`.

Attributes

Duration_t duration - specifies the length in time of the validity of the data.

Detailed Description

This `QosPolicy` specifies the duration of the validity of the data written by the `DataWriter`. When this time has expired, the data will be removed or if it has not been delivered yet, it will not be delivered at all. In other words, the duration is the time in which the data is still valid. This means that during this period a

DataReader can access the data or if the data has not been delivered yet, it still will be delivered. The default value of the `duration` is `DURATION_INFINITE`, indicating that the data does not expire.

This `QosPolicy` is applicable to a `DataWriter` and a `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` may be changed by using the `set_qos` operation.

Requested/Offered

The setting of this `QosPolicy` is only applicable to the publishing side, in other words the Requested/Offered constraints are not applicable. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side will not be raised.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.10 **LivelinessQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class LivelinessQosPolicyKind
{
 public static final LivelinessQosPolicyKind
                  AUTOMATIC_LIVELINESS_QOS;
 public static final LivelinessQosPolicyKind
                  MANUAL_BY_PARTICIPANT_LIVELINESS_QOS;
 public static final LivelinessQosPolicyKind
                  MANUAL_BY_TOPIC_LIVELINESS_QOS;
}
public final class LivelinessQosPolicy
{ public LivelinessQosPolicyKind kind;
 public Duration_t lease_duration;}
```

Description

This `QosPolicy` controls the way the liveliness of an `Entity` is being determined.

Attributes

LivelinessQosPolicyKind kind - the way the liveliness of an `Entity` is determined.

Duration_t lease_duration - the duration of the interval within which the liveliness must be determined.

Detailed Description

This `QosPolicy` controls the way the liveliness of an `Entity` is being determined. The liveliness must be reported periodically before the `lease_duration` expires.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Attributes

The `QosPolicy` is controlled by the attribute `kind` which can be:

- *AUTOMATIC_LIVELINESS_QOS* - the Data Distribution Service will take care of reporting the Liveliness automatically with a rate determined by the `lease_duration`.

- *MANUAL_BY_PARTICIPANT_LIVELINESS_QOS* - the application must take care of reporting the liveliness before the `lease_duration` expires. If an `Entity` reports its liveliness, all `Entities` within the same `DomainParticipant` that have their liveliness kind set to `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS`, can be considered alive by the Data Distribution Service. Liveliness can reported explicitly by calling the operation `assert_liveliness` on the `DomainParticipant` or implicitly by writing some data.

- *MANUAL_BY_TOPIC_LIVELINESS_QOS* - the application must take care of reporting the liveliness before the `lease_duration` expires. This can explicitly be done by calling the operation `assert_liveliness` on the `DataWriter` or implicitly by writing some data.

The `lease_duration` specifies the duration of the interval within which the liveliness should be reported.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 9 LivelinessQosPolicy

Offered \ Requested	AUTOMATIC	MANUAL_BY_PARTICIPANT	MANUAL_BY_TOPIC
AUTOMATIC	COMPATIBLE	INCOMPATIBLE	INCOMPATIBLE
MANUAL_BY_PARTICIPANT	COMPATIBLE	COMPATIBLE	INCOMPATIBLE
MANUAL_BY_TOPIC	COMPATIBLE	COMPATIBLE	COMPATIBLE

Whether communication is established, is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. In other words, the communication between any `DataWriter` and `DataReader` depends on what is expected by the `DataReader`. As a consequence, a `DataWriter` that has an incompatible `QoS` with respect to what a `DataReader` specified is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible `QoS` with respect to what a `DataWriter` specified does not get any data from that particular `DataWriter`.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.11 OwnershipQosPolicy

Scope

    DDS

Synopsis

```
import DDS.*;
public class OwnershipQosPolicyKind
{
 public static final OwnershipQosPolicyKind SHARED_OWNERSHIP_QOS;
 public static final OwnershipQosPolicyKind
      EXCLUSIVE_OWNERSHIP_QOS;
}
public final class OwnershipQosPolicy
{ public OwnershipQosPolicyKind kind; }
```

Description

This `QosPolicy` specifies whether a `DataWriter` exclusively owns an instance.

Attributes

OwnershipQosPolicyKind kind - specifies whether a `DataWriter` exclusively owns an instance.

Detailed Description

This `QosPolicy` specifies whether a `DataWriter` exclusively may own an instance. In other words, whether multiple `DataWriter` objects can write the same instance at the same time. The `DataReader` objects will only read the modifications on an instance from the `DataWriter` owning the instance.

Exclusive ownership is on an instance-by-instance basis. That is, a `Subscriber` can receive values written by a lower strength `DataWriter` as long as they affect instances whose values have not been written or registered by a higher-strength `DataWriter`.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Attribute

The `QosPolicy` is controlled by the attribute `kind` which can be:

- *SHARED_OWNERSHIP_QOS* (default) - the same instance can be written by multiple `DataWriter` objects. All updates will be made available to the `DataReader` objects. In other words it does not have a specific owner

- *EXCLUSIVE_OWNERSHIP_QOS* - the instance will only be accepted from one `DataWriter` which is the only one whose modifications will be visible to the `DataReader` objects.

Requested/Offered

In case the Requested/Offered QosPolicy are incompatible, the notification OFFERED_INCOMPATIBLE_QOS status on the offering side and REQUESTED_INCOMPATIBLE_QOS status on the requesting side is raised.

Table 10 Requested/Offered OwnershipQosPolicy

Requested Offered	**SHARED**	**EXCLUSIVE**
SHARED	compatible	INcompatible
EXCLUSIVE	INcompatible	compatible

Whether communication is established, is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. The value of the `OWNERSHIP` kind offered must exactly match the one requested or else they are considered incompatible. As a consequence, a `DataWriter` that has an incompatible `QoS` with respect to what a `DataReader` specified is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible `QoS` with respect to what a `DataWriter` specified does not get any data from that particular `DataWriter`.

Exclusive Ownership

The `DataWriter` with the highest `OwnershipStrengthQosPolicy` value and being alive (depending on the `LivelinessQosPolicy`) and which has not violated its `DeadlineQosPolicy` contract with respect to the instance, will be considered the owner of the instance. Consequently, the ownership can change as a result of:

- a `DataWriter` in the system with a higher value of the `OwnershipStrengthQosPolicy` modifies the instance

- a change in the `OwnershipStrengthQosPolicy` value (becomes less) of the `DataWriter` owning the instance

- a change in the liveliness (becomes not alive) of the `DataWriter` owning the instance

- a deadline with respect to the instance that is missed by the `DataWriter` that owns the instance

Timeline

Each `DataReader` may detect the change of ownership at a different time. In other words, at a particular point in time, the `DataReader` objects do not have a consistent picture of who owns each instance for that `Topic`. Outside this grey area in time all `DataReader` objects will consider the same `DataWriter` to be the owner.

If multiple `DataWriter` objects with the same `OwnershipStrengthQosPolicy` modify the same instance, all `DataReader` objects will make the same choice of the particular `DataWriter` that is the owner. The `DataReader` is also notified of this via a status change that is accessible by means of the `Listener` or `Condition` mechanisms.

Ownership of an Instance

`DataWriter` objects are not aware whether they own a particular instance. There is no error or notification given to a `DataWriter` that modifies an instance it does not currently own.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this `qos` by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.12 OwnershipStrengthQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class OwnershipStrengthQosPolicy
{  public int value; }
```

Description

This `QosPolicy` specifies the value of the ownership strength of a `DataWriter` used to determine the ownership of an instance.

Attributes

int value - specifies the ownership strength of the `DataWriter`.

Detailed Description

This `QosPolicy` specifies the value of the ownership strength of a `DataWriter` used to determine the ownership of an instance. This ownership is used to arbitrate among multiple `DataWriter` objects that attempt to modify the same instance. This `QosPolicy` only applies if the `OwnershipQosPolicy` is of kind `EXCLUSIVE_OWNERSHIP_QOS`. For more information, see `OwnershipQosPolicy`.

This `QosPolicy` is applicable to a `DataWriter` only. After enabling of the concerning `Entity`, this `QosPolicy` may be changed by using the `set_qos` operation. When changed, the ownership of the instances may change with it.

3.1.3.13 PartitionQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class PartitionQosPolicy
{
```

```
    public String name[];
}
```

Description

This `QosPolicy` specifies the logical `partitions` in which the `Subscribers` and `Publishers` are active.

Attributes

String name[] - the array of strings, which specifies the `partitions`.

Detailed Description

This `QosPolicy` specifies the logical `partitions` inside the domain in which the `Subscribers` and `Publishers` are active. This `QosPolicy` is particularly used to create a separate subspace, like a real domain versus a simulation domain. A `Publisher` and/or `Subscriber` can participate in more than one `partition`. Each string in the sequence of strings `name` defines a `partition` name. A `partition` name may contain wildcards. Sharing a `partition` means that at least one of the `partition` names in the sequence matches. When none of the `partition` names match, it is not considered an "incompatible" QoS and does not trigger any `listeners` or `conditions`. It only means that no communication is established. The default value of the attribute is an empty (zero-sized) sequence. This is treated as a special value that matches the "partition".

This `QosPolicy` is applicable to a `Publisher` and `Subscriber`. After enabling of the concerning `Entity`, this `QosPolicy` may be changed by using the `set_qos` operation. When changed, it modifies the association of `DataReader` and `DataWriter` objects. It may establish new associations or break existing associations. By default, `DataWriter` and `DataReader` objects belonging to a `Publisher` or `Subscriber` that do not specify a `PartitionQosPolicy`, will participate in the default `partition`. In this case the partition `name` is "".

Requested/Offered

The offered setting of this `QosPolicy` is independent of the one requested, in other words they are never considered incompatible. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side or `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side will not be raised

3.1.3.14 **PresentationQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class PresentationQosPolicyAccessScopeKind
{
 public static final PresentationQosPolicyAccessScopeKind
                INSTANCE_PRESENTATION_QOS;
 public static final PresentationQosPolicyAccessScopeKind
                TOPIC_PRESENTATION_QOS;
 public static final PresentationQosPolicyAccessScopeKind
                GROUP_PRESENTATION_QOS;
}
public final class PresentationQosPolicy
{ public PresentationQosPolicyAccessScopeKind access_scope;
  public boolean coherent_access;
  public boolean ordered_access; }
```

Description

This `QosPolicy` controls the extent to which changes to data-instances can be made dependent on each other, the order in which they need to be presented to the user and also the kind of dependencies that can be propagated and maintained by the Data Distribution Service.

Attributes

PresentationQosPolicyAccessScopeKind access_scope - specifies the granularity of the changes that needs to be preserved when communicating a set of samples and the granularity of the ordering in which these changes need to be presented to the user.

boolean coherent_access - controls whether the Data Distribution Service will preserve the groupings of changes, as indicated by the `access_scope`, made by a publishing application by means of the operations `begin_coherent_change` and `end_coherent_change`.

boolean ordered_access - controls whether the Data Distribution Service will preserve the order of the changes, as indicated by the `access_scope`.

Detailed Description

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 erroneously interpret that the aircraft is on a collision course.

Basically this QosPolicy allows a Publisher to group a number of samples by enclosing them within calls to `begin_coherent_change` and `end_coherent_change` and treat them as if they are to be communicated as a single message. 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.

Samples that belong to a (yet) unfinished coherent update consume resource limits from the receiving DataReader, but are not (yet) accessible through its history, and cannot (yet) push samples out of its history. In order for the DataReader to store samples outside its history administration, its `ResourceLimitsQosPolicy` should have a value for `max_samples_per_instance` that is bigger than the `depth` value of its `HistoryQosPolicy`.

If not enough resources are available to hold an incoming sample that belongs to an unfinished transaction, one of the following things may happen.

- When some of the resources are in use by the history of the DataReader, the incoming sample will be rejected, and the DataReader will be notified of a `SAMPLE_REJECTED` event. This will cause the delivery mechanism to retry delivery of the rejected sample at a later moment in time, in the expectation that the application will free resources by actively taking samples out of the reader history.

- When all the available resources are in use by samples belonging to unfinished coherent updates, the application has no way to free up resources and the transaction is either 'deadlocked' by itself (*i.e.* it is too big for the amount of available resources) or by one or more other incomplete transactions. To break out of this deadlock, all samples belonging to the same transaction as the currently incoming sample will be dropped, and the DataReader will be notified of a `SAMPLE_LOST` event. No attempt will be made to retransmit the dropped transaction. To avoid this scenario, it is important to make sure the DataReader has set its ResourceLimits to accommodate for the worst case history size *PLUS* the worst case transaction size. In other words, if S_h represents the worst case size of the required history, S_t represents the worst case size of single transaction and N_t represents the worst case number of concurrent transactions, then the ResourceLimits should accommodate for $S_h + (S_t * N_t)$.

A connectivity change may occur in the middle of a set of coherent changes; for example, the set of partitions used by the Publisher or one of its Subscribers may change, a late-joining DataReader 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.

PRISMTECH

The support for 'ordered_access' enables a subscribing application to view changes in the order in which they occurred. Ordering is always determined according to the applicable DestinationOrderQosPolicy setting. Depending on the selected access_scope, ordering is either on a per instance basis (this is the default behaviour, even when ordered_access is set to FALSE), on a per DataReader basis or across all DataReaders that span the Subscriber. In case of ordered_access with an access_scope of GROUP, the Subscriber will enforce that all its DataReaders share the same DestinationOrderQosPolicy setting. The DestinationOrderQosPolicy setting of the first DataReader created for that Subscriber will then determine the DestinationOrderQosPolicy setting that is allowed for all subsequent DataReaders. Conflicting settings will result in an INCONSISTENT_POLICY error.

The PresentationQosPolicy is applicable to a Publisher and Subscriber. After enabling of the concerning Entity, this QosPolicy cannot be changed any more.

Attributes

The PresentationQosPolicy is applicable to both Publisher and Subscriber, but behaves differently on the publishing side and the subscribing side. The setting of coherent_access on a Publisher controls whether that Publisher will preserve the coherency of changes (enclosed by calls to begin_coherent_change and end_coherent_change), as indicated by its access_scope and as made available by its embedded DataWriters. However, the Subscriber settings determine whether a coherent set of samples will actually be delivered to the subscribing application in a coherent way.

- If a Publisher or Subscriber sets coherent_access to FALSE, it indicates that it does not want to maintain coherency between the different samples in a set: a Subscriber that receives only a part of this set may still deliver this partial set of samples to its embedded DataReaders.

- If both Publisher and Subscriber set coherent_access to TRUE, they indicate that they want to maintain coherency between the different samples in a set: a Subscriber that receives only a part of this set may not deliver this partial set of samples to its embedded DataReaders; it needs to wait for the set to become complete, and it will flush this partial set when it concludes that it will never be able to complete it.

  Coherency is implemented on top of a transaction mechanism between individual DataWriters and DataReaders; completeness of a coherent set is determined by the successful completion of each of its participating transactions. The value of the access_scope attribute determines which combination of transactions constitute the contents of a coherent set.

The setting of `ordered_access` has no impact on the way in which a Publisher transmits its samples (although it does influence the RxO properties of this Publisher), but basically it determines whether a Subscriber will preserve the ordering of samples when the subscribing application uses its embedded DataReaders to read or take samples:

- If a Subscriber sets `ordered_access` to `FALSE`, it indicates that it does not want to maintain ordering between the different samples it receives: a subscribing application that reads or takes samples will receive these samples ordered by their key-values, which does probably not resemble the order they were written in.

- If a Subscriber sets `ordered_access` to `TRUE`, it indicates that it does want to maintain ordering within the specified `access_scope` between the different samples it receives: a subscribing application that reads or takes samples will receives these samples sorted by the order in which they were written.

The `access_scope` determines the maximum extent of coherent and/or ordered changes:

- If `access_scope` is set to `INSTANCE_PRESENTATION_QOS` and `coherent_access` is set to `TRUE`, then the Subscriber will behave, with respect to maintaining coherency, in a way similar to an `access_scope` that is set to `TOPIC_PRESENTATION_QOS`. This is caused by the fact that coherency is defined as the successful completion of all participating transactions. If a DataWriter writes a transaction containing samples from different instances, and a connected DataReader misses one of these samples, then the transaction failed and the coherent set is considered incomplete by the receiving DataReader. It doesn't matter that all the other instances have received their samples successfully; an unsuccessful transaction by definition results in an incomplete coherent set. In that respect the DDS can offer no granularity that is more fine-grained with respect to coherency than that described by the `TOPIC_PRESENTATION_QOS`.

  If `access_scope` is set to `INSTANCE_PRESENTATION_QOS` and `ordered_access` is set to `TRUE`, then the subscriber will maintain ordering between samples belonging to the same instance. Samples belonging to different instances will still be grouped by their key-values instead of by the order in which they were received.

- If `access_scope` is set to `TOPIC_PRESENTATION_QOS` and `coherent_access` is set to `TRUE`, then the DDS will define the scope of a coherent set on individual transactions. So a coherent set that spans samples coming from multiple DataWriters (indicated by its enclosure within calls to `begin_coherent_change` and `end_coherent_change` on their shared Publisher), is chopped up into separate and disjunct transactions (one for each participating DataWriter), where each transaction is processed separately. On the subscribing side this may result in the successful completion of some of these

transactions, and the unsuccessful completion of some others. In such cases all DataReaders that received successful transactions will deliver the embedded content to their applications, without waiting for the completion of other transactions in other DataReaders connected to the same Subscriber.

If `access_scope` is set to `TOPIC_PRESENTATION_QOS` and `ordered_access` is set to `TRUE`, then the subscriber will maintain ordering between samples belonging to the same DataReader. This means that samples belonging to the same instance in the same DataReader may no longer be received consecutively if samples belonging to different instances were written in between. It is possible to read/take a limited batch of ordered samples (where `max_samples  !=  LENGTH_UNLIMITED`). In that case the DataReader will keep a bookmark, so that in subsequent read/take operations your application can start where the previous read/take call left off. There are two ways for the middleware to indicate that you completed a full iteration:

- The amount of samples returned is smaller than the amount of samples requested.

- Your read/take call returns `RETCODE_NO_DATA`, indicating that there are no more samples matching your criteria after the current bookmark. In that case the bookmark is reset and the next read/take will begin a new iteration right from the start of the ordered list.

The bookmark is also reset in the following cases:

- A read/take call uses different masks than the previous invocation of read/take.

- A read/take call uses a different query than the previous invocation of read/take.

• If   `access_scope`   is   set   to   `GROUP_PRESENTATION_QOS`   and `coherent_access` is set to `TRUE`, then the DDS will define the scope of a coherent set on the sum of all participating transactions. So a coherent set that spans samples coming from multiple DataWriters (indicated by its enclosure within calls to `begin_coherent_change` and `end_coherent_change` on their shared Publisher), is chopped up into separate and disjunct transactions (one for each participating DataWriter), where each transactions is processed separately. On the subscribing side this may result in the successful completion of some of these transactions, and the unsuccessful completion of some others. However, each DataReader is only allowed to deliver the embedded content when all participating transactions completed successfully. This means that DataReaders that received successful transactions will need to wait for all other DataReaders attached to the same Subscriber to also complete their transactions successfully. If one or more DataReaders conclude that they will not be able to complete their transactions successfully, then all DataReaders that participate in the original coherent set will flush the content of their transactions. In order for the application to access the state of all DataReaders that span the coherent

update, a separate read/take operation will need to be performed on each of the concerned DataReaders. To keep the history state of the DataReaders consistent in between the successive invocations of the read/take operations on the various readers, the DataReaders should be locked for incoming updates by invoking the `begin_access` on the Subscriber prior to accessing the first DataReader. If all concerned DataReaders have been accessed properly, they can be unlocked for incoming updates by invoking the `end_access` on the Subscriber.

If `access_scope` is set to `GROUP_PRESENTATION_QOS` and `ordered_access` is set to `TRUE`, then ordering is maintained between samples that are written by DataWriters attached to a common Publisher and received by DataReaders attached to a common Subscriber. This way the subscribing application can access the changes as a unit and/or in the proper order. However, this does not necessarily imply that the subscribing application will indeed access the changes as a unit and/or in the correct order. For that to occur, the subscribing application must use the proper logic in accessing its datareaders:

- Upon notification by the callback operation `on_data_on_readers` of the SubscriberListener or when triggered by the similar `DATA_ON_READERS` status of the Subscriber's `StatusCondition`, the application uses `begin_access` on the Subscriber to indicate it will be accessing data through the Subscriber. This will lock the embedded datareaders for any incoming messages during the coherent data access.

- Then it calls `get_datareaders` on the Subscriber to get the list of DataReader objects where data samples are available. Note that when `ordered_access` is `TRUE`, then the list of DataReaders may contain the same reader several times. In this manner the correct sample order can be maintained among samples in different DataReader objects.

- Following this it calls `read` or `take` on each DataReader in the same order returned to access all the relevant changes in the DataReader. Note that when `ordered_access` is `TRUE`, you should only read or take *one* sample at a time.

- Once it has called `read` or `take` on all the readers, it calls `end_access` on the Subscriber. This will unlock the embedded datareaders again.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

PRISMTECH

Table 11 Requested/Offered PresentationQosPolicy

Requested / Offered	INSTANCE	Topic	Group
instance	compatible	INcompatible	INcompatible
topic	compatible	compatible	INcompatible
group	compatible	compatible	compatible

The value offered is considered compatible with the value requested if and only if the following conditions are met:

1. The inequality "offered `access_scope` >= requested `access_scope`" evaluates to 'TRUE'. For the purposes of this inequality, the values of `PRESENTATION access_scope` are considered ordered such that `INSTANCE < TOPIC < GROUP`.

2. Requested `coherent_access` is `FALSE`, or else both offered and requested `coherent_access` are `TRUE`.

3. Requested `ordered_access` is `FALSE`, or else both offered and requested `ordered_access` are `TRUE`.

In case the quality offered by the Publisher is better than the value requested by the Subscriber, the subscriber's values determine the resulting behaviour for the subscribing application. In other words, the quality specified at the Subscriber site overrules the corresponding value at the Publisher site.

Consider the following scenario:

1. A Publisher publishes coherent sets with `access_scope` is `GROUP` and `coherent_access` is `TRUE`.

2. A Subscriber subscribes to these coherent sets with `access_scope` is `TOPIC` and `coherent_access` is `TRUE`.

3. The Publisher writes a coherent set consisting of 2 samples of Topic A, and 2 samples of Topic B.

4. During transmission, the first sample of Topic B gets lost.

According to the `access_scope` of the Publisher, the coherent set is incomplete and can therefore not be delivered. However, according to the `access_scope` of the Subscriber, coherency needs to be maintained on a per Reader/Writer pair basis so the samples for Topic A will be delivered upon arrival, but the samples for Topic B will not.

Basically, when both `coherent_access` and `ordered_access` are set to `FALSE`, then the `access_scope` serves no other purpose than to determine connectivity between Publishers and Subscribers.

3.1.3.15 ReaderDataLifecycleQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class InvalidSampleVisibilityQosPolicyKind
{
  public static final InvalidSampleVisibilityQosPolicyKind
              NO_INVALID_SAMPLES;
  public static final InvalidSampleVisibilityQosPolicyKind
              MINIMUM_INVALID_SAMPLES;
  public static final InvalidSampleVisibilityQosPolicyKind
              ALL_INVALID_SAMPLES;
}
public final class InvalidSampleVisibilityQosPolicy
{
  public InvalidSampleVisibilityQosPolicyKind kind;
}
public final class ReaderDataLifecycleQosPolicy
{
  public Duration_t autopurge_nowriter_samples_delay;
  public Duration_t autopurge_disposed_samples_delay;
  public boolean autopurge_dispose_all;
  public boolean enable_invalid_samples;
  public InvalidSampleVisibilityQosPolicy
              invalid_sample_visibility;
}
```

Description

This `QosPolicy` specifies the maximum duration for which the DataReader will maintain information regarding a data instance for which the instance_state becomes either `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`.

Attributes

Duration_t autopurge_nowriter_samples_delay - specifies the duration for which the `DataReader` will maintain information regarding a data instance for which the instance_state becomes `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`. By default the duration value is `DURATION_INFINITE`. When the delay time has expired, the data instance is marked so that it can be purged in the next garbage collection sweep.

PRISMTECH

Duration_t autopurge_disposed_samples_delay - specifies the duration for which the `DataReader` will maintain information regarding a data instance for which the `instance_state` becomes `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. By default the duration value is `DURATION_INFINITE`. When the delay time has expired, the data instance is marked so that it can be purged in the next garbage collection sweep.

Boolean autopurge_dispose_all - Determines whether all samples in the `DataReader` will be purged automatically when a `dispose_all_data()` call is performed on the `Topic` that is associated with the `DataReader`. If this attribute set to `TRUE`, no more samples will exist in the `DataReader` after the `dispose_all_data` has been processed. Because all samples are purged, no data available events will be notified to potential Listeners or Conditions that are set for the `DataReader`. If this attribute is set to `FALSE`, the `dispose_all_data()` behaves as if each individual instance was disposed separately.

Boolean enable_invalid_samples - Insert dummy samples if no data sample is available to notify readers of an instance state change. By default the value is `TRUE`.

⚠️ **NOTE:** This feature is deprecated. It is recommended that you use `invalid_sample_visibility` instead.

InvalidSampleVisibilityQosPolicy invalid_sample_visibility - Insert dummy samples if no data sample is available, to notify readers of an instance state change. By default the value is `MINIMUM_INVALID_SAMPLES`.

Detailed Description

This `QosPolicy` specifies the maximum duration for which the `DataReader` will maintain information regarding a data instance for which the `instance_state` becomes either `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. The `DataReader` manages resources for instances and samples of those instances. The amount of resources managed depends on other `QosPolicies` like the `HistoryQosPolicy` and the `ResourceLimitsQosPolicy`. The `DataReader` can only release resources for data instances for which all samples have been taken and the `instance_state` has become `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. If an application does not take the samples belonging to a data instance with such an `instance_state`, the `DataReader` will never be able to release the maintained resources. By means of this `QosPolicy` the application can instruct the `DataReader` to release all resources related to the concerning data instance after a specified duration.

⚠️ There is one exception to this rule. If the `autopurge_dispose_all` attribute is `TRUE`, the maintained resources in the `DataReader` are cleaned up immediately in case `dispose_all_data()` is called on the `Topic` that is associated with the `DataReader`.

Instance state changes are communicated to a `DataReader` by means of the `SampleInfo` accompanying a data sample. If no samples are available in the `DataReader`, a so-called 'invalid sample' can be injected with the sole purpose of notifying applications of the instance state. This behaviour is configured by the `InvalidSampleVisibilityQosPolicy`.

- If `invalid_sample_visibility` is set to `NO_INVALID_SAMPLES`, applications will be notified of instance_state changes only if there is a sample available in the `DataReader`. The `SampleInfo` belonging to this sample will contain the updated instance state.

- If `invalid_sample_visibility` is set to `MINIMUM_INVALID_SAMPLES`, the middleware will try to update the `instance_state` on available samples in the `DataReader`. If no sample is available, an invalid sample will be injected. These samples contain only the key values of the instance. The `SampleInfo` for invalid samples will have the 'valid_data' flag disabled, and contain the updated instance state.

- If `invalid_sample_visibility` is set to `ALL_INVALID_SAMPLES`, every change in the `instance_state` will be communicated by a separate invalid sample.

⚠️ **NOTE:** This value (`ALL_INVALID_SAMPLES`) is not yet implemented. It is scheduled for a future release.

An alternative but deprecated way to determine the visibility of state changes is to set a boolean value for the `enable_invalid_samples` field.

- When `TRUE`, the behavior is similar to the `MINIMUM_INVALID_SAMPLES` value of the `InvalidSampleVisibilityQosPolicy` field.

- When `FALSE`, the behavior is similar to the `NO_INVALID_SAMPLES` value of the `InvalidSampleVisibilityQosPolicy` field.

⚠️ You cannot set both the the `enable_invalid_samples` field *AND* the `invalid_sample_visibility` field. If both deviate from their factory default, this is considered a `RETCODE_INCONSISTENT_POLICY`. If only one of the fields deviates from its factory default, then that setting will be leading. However, modifying the default value of the `enable_invalid_samples` field will automatically result in a warning message stating that you are using deprecated functionality.

This `QosPolicy` is applicable to a `DataReader` only. After enabling the relevant `DataReader`, this `QosPolicy` can be changed using the `set_qos` operation.

3.1.3.16 **ReliabilityQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class ReliabilityQosPolicyKind
{
 public static final ReliabilityQosPolicyKind
                 BEST_EFFORT_RELIABILITY_QOS;
 public static final ReliabilityQosPolicyKind
                 RELIABLE_RELIABILITY_QOS;
}
public final class ReliabilityQosPolicy
{
 public ReliabilityQosPolicyKind kind;
 public Duration_t max_blocking_time;
 public boolean synchronous;
}
```

Description

This `QosPolicy` controls the level of reliability of the data distribution offered or requested by the `DataWriters` and `DataReaders`.

Attributes

ReliabilityQosPolicyKind kind - specifies the type of reliability which may be `BEST_EFFORT_RELIABILITY_QOS` or `RELIABLE_RELIABILITY_QOS`.

Duration_t max_blocking_time - specifies the maximum time the `write` operation may block when the `DataWriter` does not have space to store the value or when synchronous communication is specified and all expected acknowledgements are not yet received.

boolean synchronous - specifies whether a `DataWriter` should wait for acknowledgements by all connected `DataReaders` that also have set a synchronous `ReliabilityQosPolicy`.

It is advisable only to use this policy in combination with `RELIABLE_RELIABILITY`; if used in combination with `BEST_EFFORT` data may not arrive at the `DataReader`, resulting in a timeout at the `DataWriter` indicating that the data has not been received.

Acknowledgments are always sent `RELIABLE` so that when the `DataWriter` encounters a timeout it is guaranteed that the `DataReader` hasn't received the data.

⚠️ **NOTE:** This is an OpenSplice-specific parameter, it is *not* part of the DDS Specification.

Detailed Description

This `QosPolicy` controls the level of reliability of the data distribution requested by a `DataReader` or offered by a `DataWriter`. In other words, it controls whether data is allowed to get lost in transmission or not.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Attributes

- *RELIABLE_RELIABILITY_QOS* - the Data Distribution Service will attempt to deliver all samples in the `DataWriters` history; arrival-checks are performed and data may get re-transmitted in case of lost data. In the steady-state (no modifications communicated via the `DataWriter`) the Data Distribution Service guarantees that all samples in the `DataWriter` history will eventually be delivered to the all `DataReader` objects. Outside the steady-state the `HistoryQosPolicy` and `ResourceLimitsQosPolicy` determine how samples become part of the history and whether samples can be discarded from it. In this case also the `max_blocking_time` must be set

- *BEST_EFFORT_RELIABILITY_QOS* - the Data Distribution Service will only attempt to deliver the data; no arrival-checks are being performed and any lost data is not re-transmitted (non-reliable). Presumably new values for the samples are generated often enough by the application so that it is not necessary to resent or acknowledge any samples.

The effect of the attribute `max_blocking_time` depends on the setting of the `HistoryQosPolicy` and `ResourcesLimitsQosPolicy` and/or the synchronous setting of the `ReliabilityQosPolicy`. In case the `HistoryQosPolicy` kind is set to `KEEP_ALL_HISTORY_QOS`, the `write` operation on the `DataWriter` may block if the modification would cause one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. Also in case the synchronous attribute value of the `ReliabilityQosPolicy` is set to `TRUE` on both sides of a pair of connected `DataWriters` and `DataReaders`, then the `DataWriter` will wait until all its connected synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum duration the `write` operation may block.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 12 Requested/Offered ReliabilityQosPolicy

Requested Offered	**BEST_EFFORT**	**RELIABLE**
BEST_EFFORT	compatible	INcompatible
RELIABLE	compatible	compatible

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.17 ResourceLimitsQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class ResourceLimitsQosPolicy
{
 public int max_samples;
 public int max_instances;
 public int max_samples_per_instance;
}
```

Description

This `QosPolicy` will specify the maximum amount of resources, which can be used by a `DataWriter` or `DataReader`.

Attributes

int max_samples - the maximum number of data samples for all instances for any single `DataWriter` (or `DataReader`). By default, LENGTH_UNLIMITED.

int max_instances - the maximum number of instances for any single `DataWriter` (or `DataReader`). By default, LENGTH_UNLIMITED.

int max_samples_per_instance - the maximum number of samples of any single instance for any single `DataWriter` (or `DataReader`). By default, LENGTH_UNLIMITED.

Detailed Description

This `QosPolicy` controls the maximum amount of resources that the Data Distribution Service can use in order to meet the requirements imposed by the application and other `QosPolicy` settings.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Requested/Offered

The value of the `QosPolicy` offered is independent of the one requested, in other words they are never considered incompatible. The communication will not be rejected on account of this `QosPolicy`. The notification OFFERED_INCOMPATIBLE_QOS status on the offering side or REQUESTED_INCOMPATIBLE_QOS status on the requesting side will not be raised.

Resource Limits

If the `DataWriter` objects are publishing samples faster than they are taken by the `DataReader` objects, the Data Distribution Service will eventually hit against some of the `QosPolicy`-imposed resource limits. Note that this may occur when just a single `DataReader` cannot keep up with its corresponding `DataWriter`.

In case the `HistoryQosPolicy` is KEEP_LAST_HISTORY_QOS, the setting of `ResourceLimitsQosPolicy max_samples_per_instance` must be compatible with the `HistoryQosPolicy` depth. For these two `QosPolicy` settings to be compatible, they must verify that depth <= max_samples_per_instance.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.18 SchedulingQosPolicy

⚠ **NOTE:** This is an OpenSplice-specific QosPolicy, it is *not* part of the DDS Specification.

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class SchedulingClassQosPolicyKind
{
 public static final SchedulingClassQosPolicyKind
                 SCHEDULE_DEFAULT;
 public static final SchedulingClassQosPolicyKind
                 SCHEDULE_TIMESHARING;
 public static final SchedulingClassQosPolicyKind
                 SCHEDULE_REALTIME;
}
public class SchedulingClassQosPolicy
{
 public SchedulingClassQosPolicyKind kind;
}
public class SchedulingPriorityQosPolicyKind
{
 public static final SchedulingPriorityQosPolicyKind
                 PRIORITY_RELATIVE;
 public static final SchedulingPriorityQosPolicyKind
                 PRIORITY_ABSOLUTE;
}
public class SchedulingPriorityQosPolicy
{
 public SchedulingPriorityQosPolicyKind kind;
}
public class SchedulingQosPolicy
{
 public SchedulingClassQosPolicy scheduling_class;
 public SchedulingPriorityQosPolicy scheduling_priority_kind;
 public int scheduling_priority;
}
```

Description

This `QosPolicy` specifies the scheduling parameters that will be used for a thread that is spawned by the `DomainParticipant`.

⚠ Note that some scheduling parameters may not be supported by the underlying Operating System, or that you may need special privileges to select particular settings.

Attributes

SchedulingClassQosPolicyKind scheduling_class.kind - specifies the
scheduling class used by the Operating System, which may be
SCHEDULE_DEFAULT, SCHEDULE_TIMESHARING or SCHEDULE_REALTIME.
Threads can only be spawned within the scheduling classes that are supported
by the underlying Operating System.

SchedulingPriorityQosPolicyKind scheduling_priority_kind.kind
- specifies the priority type, which may be either PRIORITY_RELATIVE or
PRIORITY_ABSOLUTE.

int scheduling_priority - specifies the priority that will be assigned to threads
spawned by the DomainParticipant. Threads can only be spawned with
priorities that are supported by the underlying Operating System.

Detailed Description

This QosPolicy specifies the scheduling parameters that will be used for threads
spawned by the DomainParticipant. Note that some scheduling parameters may
not be supported by the underlying Operating System, or that you may need special
privileges to select particular settings. Refer to the documentation of your OS for
more details on this subject.

Although the behaviour of the scheduling_class is highly dependent on the
underlying OS, in general it can be said that when running in a Timesharing class
your thread will have to yield execution to other threads of equal priority regularly.
In a Realtime class your thread normally runs until completion, and can only be
pre-empted by higher priority threads. Often the highest range of priorities is not
accessible through a Timesharing Class.

The scheduling_priority_kind determines whether the specified
scheduling_priority should be interpreted as an absolute priority, or whether it
should be interpreted relative to the priority of its creator, in this case the priority of
the thread that created the DomainParticipant.

3.1.3.19 TimeBasedFilterQosPolicy

Scope

DDS

Synopsis

```
import DDS.*;
public final class TimeBasedFilterQosPolicy
{
 public Duration_t minimum_separation;
}
```

PRISMTECH

Description

This `QosPolicy` specifies a period after receiving a sample for a particular instance during which a `DataReader` will filter out new samples for the same instance.

At the end of the period the latest state of the instance will be notified and a new filter period will start. If there are no new samples in a period the filter will *not* notify the same latest already-notified state and it will wait for a new sample on this particular instance to start a new period.

In the case where the reliability QoS kind is `RELIABLE` the system guarantees that the latest state is notified.

Effectively the `DataReader` will receive at most one sample with the latest state *per* period for each instance.

Attributes

Duration_t minimum_separation – specifies the minimum period between received samples to be passed through the filter. The default value is `0`, meaning that all samples are accepted.

Detailed Description

This `QosPolicy` allows a `DataReader` to indicate that it is not interested in processing all samples for each instance. Instead it requests at most one change per `minimum_separation` period.

The filter is applied to each data-instance separately. This means that new instances will not be filtered, no matter what the `minimum_separation` period or their publication time is. The filter is only applied to samples belonging to the same instance, limiting the rate at which the `DataReader` is notified of the most current value of each instance. This can be helpful in situations where some nodes are capable of generating data much faster than others can consume it. Instance state changes are not affected by the filter, so a `DataReader` always contains the latest state of an instance.

The `minimum_separation` period must be consistent with the `DeadlineQosPolicy`. If the `minimum_separation` period is greater than the deadline period, the deadline cannot be met; therefore the two QoS policies are inconsistent. An attempt to set these policies with inconsistent values will result in a failure to create the `DataReader` or an `INCONSISTENT_POLICY` return value.

This `QosPolicy` is applicable to a `DataReader` only. After enabling the relevant `DataReader`, this `QosPolicy` can be changed using the `set_qos` operation.

3.1.3.20 **TopicDataQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class TopicDataQosPolicy
{
 public byte value[];
}
```

Description

This `QosPolicy` allows the application to attach additional information to a Topic Entity. This information is distributed with the BuiltinTopics.

Attributes

byte value[] – a sequence of bytes that holds the application topic data. By default, the sequence has length 0.

Detailed Description

This `QosPolicy` allows the application to attach additional information to a `Topic` Entity. This information is distributed with the `BuiltinTopic`. An application that discovers a new `Topic` entity, can use this information to add additional functionality. The `TopicDataQosPolicy` is changeable and updates of the `BuiltinTopic` instance must be expected. Note that the Data Distribution Service is not aware of the real structure of the topic data (the Data Distribution System handles it as an opaque type) and that the application is responsible for correct mapping on structural types for the specific platform.

3.1.3.21 **TransportPriorityQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class TransportPriorityQosPolicy
{
 public int value;
}
```

Description

This `QosPolicy` specifies the priority with which the Data Distribution System can handle the data produced by the `DataWriter`.

Attributes

`int value` – specifies the priority with which the Data Distribution System can handle the data produced by the `DataWriter`.

Detailed Description

This `QosPolicy` specifies the priority with which the Data Distribution System can handle the data produced by a `DataWriter`. This `QosPolicy` is considered to be a hint to the Data Distribution Service to control the priorities of the underlying transport means. A higher value represents a higher priority and the full range of the type is supported. By default the transport priority is set to 0.

The `TransportPriorityQosPolicy` is applicable to both `Topic` and `DataWriter` entities. After enabling of the concerning `Entities`, this `QosPolicy` may be changed by using the `set_qos` operation.

TopicQos

Note that changing this `QosPolicy` for the `Topic` does not influence the behaviour of the Data Distribution System for existing `DataWriter` entities because this `QosPolicy` is only used by the operation `copy_from_topic_qos` and when specifying `DATAWRITER_QOS_USE_TOPIC_QOS` when creating the `DataWriter`.

3.1.3.22 UserDataQosPolicy

Scope

DDS

Synopsis

```
import DDS.*;
public final class UserDataQosPolicy
{
 public byte value[];
}
```

Description

This `QosPolicy` allows the application to attach additional information to a `DomainParticipant`, `DataReader` or `DataWriter` entity. This information is distributed with the Builtin Topics.

Attributes

byte value[] – a sequence of bytes that holds the application user data. By default, the sequence has length 0.

Detailed Description

This `QosPolicy` allows the application to attach additional information to a `DomainParticipant`, `DataReader` or `DataWriter` entity. This information is distributed with the Builtin Topics. An application that discovers a new `Entity` of the listed kind, can use this information to add additional functionality. The `UserDataQosPolicy` is changeable and updates of the Builtin Topic instance must be expected. Note that the Data Distribution Service is not aware of the real structure of the user data (the Data Distribution System handles it as an opaque type) and that the application is responsible for correct mapping on structural types for the specific platform.

3.1.3.23 WriterDataLifecycleQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class WriterDataLifecycleQosPolicy
{
 public boolean autodispose_unregistered_instances;
 public Duration_t autopurge_suspended_samples_delay;
 public Duration_t autounregister_instance_delay;
}
```

Description

This `QosPolicy` drives the behaviour of a DataWriter concerning the life-cycle of the instances and samples that have been written by it.

Attributes

boolean autodispose_unregistered_instances – specifies whether the Data Distribution Service should automatically dispose instances that are unregistered by this `DataWriter`. By default this value is `true`.

Duration_t autopurge_suspended_samples_delay – specifies the duration after which the `DataWriter` will automatically remove a sample from its history during periods in which its Publisher is suspended. This duration is calculated based on the source timestamp of the written sample. By default the

PRISMTECH

duration value is set to `DURATION_INFINITE` and therefore no automatic purging of samples occurs. See Section 3.4.1.19, *suspend_publications*, on page 273 for more information about suspended publication.

Duration_t autounregister_instance_delay – specifies the duration after which the `DataWriter` will automatically unregister an instance after the application wrote a sample for it and no further action is performed on the same instance by this `DataWriter` afterwards. This means that when the application writes a new sample for this instance, the duration is recalculated from that action onwards. By default the duration value is `DURATION_INFINITE` and therefore no automatic unregistration occurs.

Detailed Description

This `QosPolicy` controls the behaviour of the `DataWriter` with regards to the lifecycle of the data-instances it manages, that is, the data instances that have been registered either explicitly using one of the `register` operations or implicitly by directly writing the data using the special `HANDLE_NIL` parameter. (See also Section 3.4.2.50, *register_instance*, on page 315).

The `autodispose_unregistered_instances` flag controls what happens when an instance gets unregistered by the `DataWriter`:

- If the `DataWriter` unregisters the instance explicitly using either `unregister_instance` or `unregister_instance_w_timestamp`, then the `autodispose_unregistered_instances` flag is currently ignored and the instance is never disposed automatically.

- If the `DataWriter` unregisters its instances implicitly because it is deleted, or if a `DataReader` detects a loss of liveliness of a connected `DataWriter`, or if `autounregister_instance_delay` expires, then the `autodispose_unregistered_instances` flag determines whether the concerned instances are automatically disposed (`true`) or not (`false`).

For `DataWriters` associated with `TRANSIENT` and `PERSISTENT` topics setting the `autodispose_unregister_instances` attribute to `true` would mean that all instances that are not explicitly unregistered by the application will by default be removed from the Transient and Persistent stores when the DataWriter is deleted, when a loss of liveliness is detected, or when the `autounregister_instance_delay` expires.

3.1.3.24 SubscriptionKeyQosPolicy

Scope

DDS

Synopsis

```
import DDS.*;
public class SubscriptionKeyQosPolicy
{   public boolean use_key_list;
    public String[] key_list;   };
```

Description

This QosPolicy allows the DataReader to define it's own set of keys on the data, potentially different from the keys defined on the topic.

⚠ **NOTE:** This is an OpenSplice-specific QosPolicy, it is *not* part of the DDS Specification.

Attributes

`boolean use_key_list` – Controls whether the alternative key list is applied on the DataReader.

`String[] key_list` – A sequence of strings with one or more names of topic fields acting as alternative keys.

Detailed Description

By using the SubscriptionKeyQosPolicy, a DataReader can force its own key-list definition on data samples. The consequences are that the DataReader will internally keep track of instances based on its own key list, instead of the key list dictated by the Topic.

Operations that operate on instances or instance handles, such as `lookup_instance` or `get_key_value`, respect the alternative key-list and work as expected. However, since the mapping of writer instances to reader instances is no longer trivial (one writer instance may now map to more than one matching reader instance and *vice versa*), a writer instance will no longer be able to fully determine the lifecycle of its matching reader instance, nor the value its `view_state` and `instance_state`.

In fact, by diverting from the conceptual 1 – 1 mapping between writer instance and reader instance, the writer can no longer keep an (empty) reader instance `ALIVE` by just refusing to unregister its matching writer instance. That means that when a reader takes all samples from a particular reader instance, that reader instance will immediately be removed from the reader's administration. Any subsequent reception of a message with the same keys will re-introduce the instance into the reader administration, setting its `view_state` back to `NEW`. Compare this to the default behaviour, where the reader instance will be kept alive as long as the writer does not unregister it. That causes the `view_state` in the reader instance to remain `NOT_NEW`, even if the reader has consumed all of its samples prior to receiving an update.

PRISMTECH

Another consequence of allowing an alternative keylist is that events that are communicated by invalid samples (*i.e.* samples that have only initialized their keyfields) may no longer be interpreted by the reader to avoid situations in which uninitialized non-keyfields are treated as keys in the alternative keylist. This effectively means that all invalid samples (*e.g.* unregister messages and both implicit and explicit dispose messages) will be skipped and can no longer affect the `instance_state`, which will therefore remain `ALIVE`. The only exceptions to this are the messages that are transmitted explicitly using the `writedispose()` call (see Section 3.4.2.59, *writedispose*, on page 327), which always includes a full and valid sample and can therefore modify the `instance_state` to `NOT_ALIVE_DISPOSED`.

By default, the SubscriptionKeyQosPolicy is not used because `use_key_list` is set to `FALSE`.

This QosPolicy is applicable to a DataReader only, and cannot be changed after the DataReader is enabled.

3.1.3.25 ReaderLifespanQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class ReaderLifespanQosPolicy
{   public boolean use_lifespan;
     public Duration_t duration;   };
```

Description

Automatically remove samples from the DataReader after a specified timeout.

⚠️ **NOTE:** This is an OpenSplice-specific QosPolicy, it is *not* part of the DDS Specification.

Attributes

boolean use_lifespan - Controls whether the lifespan is applied to the samples in the DataReader.

Duration_t duration - The duration after which data loses validity and is removed.

Detailed Description

This QosPolicy is similar to the `LifespanQosPolicy` (applicable to Topic and DataWriter), but limited to the DataReader on which the QosPolicy is applied. The data is automatically removed from the DataReader if it has not been taken yet after the lifespan duration expires. The duration of the ReaderLifespan is added to the insertion time of the data in the DataReader to determine the expiry time.

When both the `ReaderLifespanQosPolicy` and a DataWriter's `LifespanQosPolicy` are applied to the same data, only the earliest expiry time is taken into account.

By default, the ReaderLifespanQosPolicy is not used and `use_lifespan` is `FALSE`. The duration is set to `DURATION_INFINITE`.

This QosPolicy is applicable to a DataReader only, and is mutable even when the DataReader is already enabled. If modified, the new setting will only be applied to samples that are received after the modification took place.

3.1.3.26 ShareQosPolicy

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class ShareQosPolicy
{   public boolean enable;
     public String name; };
```

Description

Used to share a DataReader between multiple processes.

⚠ **NOTE:** This is an OpenSplice-specific QosPolicy, it is *not* part of the DDS Specification.

Attributes

boolean enable - Controls whether the entity is shared.

String name - The label used to identify the shared Entity.

Detailed Description

This QosPolicy allows sharing of entities by multiple processes or threads. When the policy is enabled, the data distribution service will try to look up an existing entity that matches the name supplied in the ShareQosPolicy. A new entity will only be created if a shared entity registered under the specified name doesn't exist yet.

Shared Readers can be useful for implementing algorithms like the worker pattern, where a single shared reader can contain samples representing different tasks that may be processed in parallel by separate processes. In this algorithm each processes consumes the task it is going to perform (*i.e.* it takes the sample represening that task), thus preventing other processes from consuming and therefore performing the same task.

NOTE: Entities can only be shared between processes if OpenSplice is running in federated mode, because it requires shared memory to communicate between the different processes.

By default, the ShareQosPolicy is not used and `enable` is `FALSE`. `Name` must be set to a valid string for the ShareQosPolicy to be valid when `enable` is set to `TRUE`.

This QosPolicy is applicable to DataReader and Subscriber entities, and cannot be modified after the DataReader or Subscriber is enabled. Note that a DataReader can only be shared if its Subscriber is also shared.

3.1.3.27 **ViewKeyQosPolicy**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class ViewKeyQosPolicy
{   public boolean use_key_list;
    public String[] key_list;   };
```

Description

Used to define a set of keys on a DataReaderView.

NOTE: This is an OpenSplice-specific QosPolicy, it is *not* part of the DDS Specification.

Detailed Description

This QosPolicy is used to set the key list of a DataReaderView. A DataReaderView allows a different view, defined by this key list, on the data set of the DataReader from which it is created.

Operations that operate on instances or instance handles, such as `lookup_instance` or `get_key_value`, respect the alternative key-list and work as expected. However, since the mapping of writer instances to reader instances is no longer trivial (one writer instance may now map to more than one matching

reader instance and *vice versa*), a writer instance will no longer be able to fully determine the lifecycle of its matching reader instance, nor the value its `view_state` and `instance_state`.

In fact, the view sample will always copy the `view_state` and `instance_state` values from the reader sample to which it is slaved. If both samples preserve a $1 - 1$ correspondence with respect to their originating instances (this may sometimes be the case even when an alternative keylist is provided, *i.e.* when one reader instance never maps to more than one view instance and *vice versa*) then the resulting `instance_state` and `view_state` still have a valid semantical meaning. If this $1 - 1$ correspondence cannot be guaranteed, the resulting `instance_state` and `view_state` are semantically meaningless and should not be used to derive any conclusion regading the lifecycle of a view instance.

By default, the `ViewKeyQosPolicy` is disabled because `use_key_list` is set to `FALSE`.
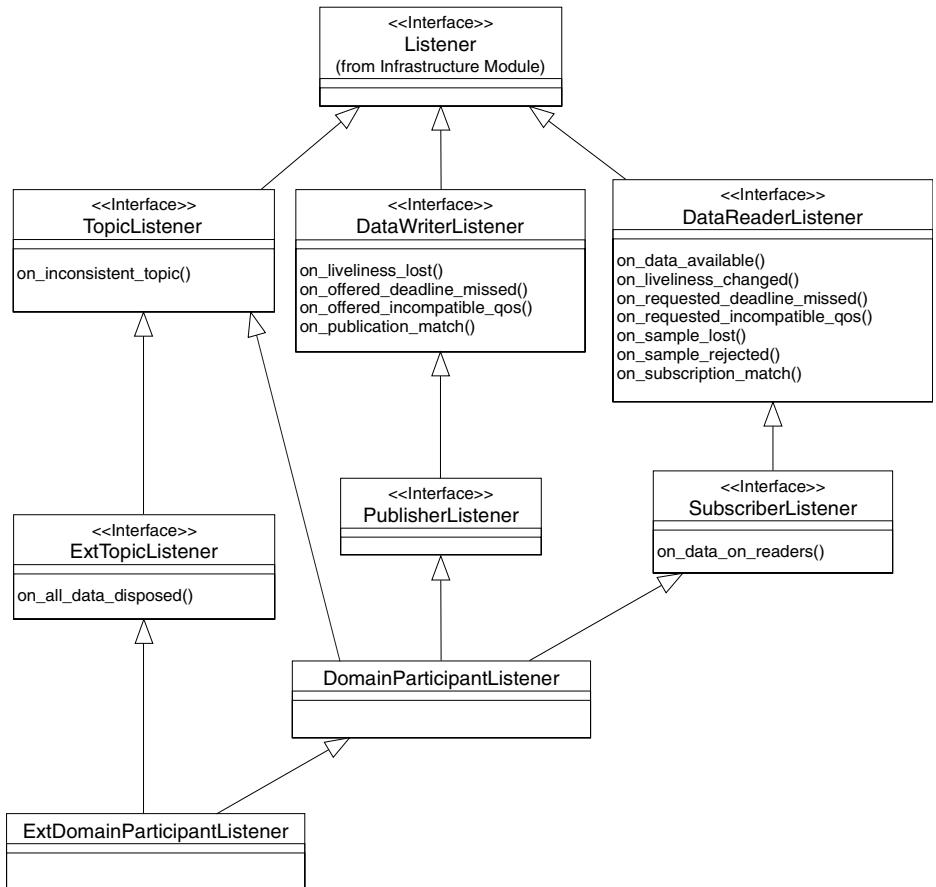
This QosPolicy is applicable to a DataReaderView only, and cannot be changed after the DataReaderView is created.

3.1.4 Listener Interface

This interface is the abstract base interface for all `Listener` interfaces. `Listeners` provide a generic mechanism for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. Each DCPS `Entity` supports its own specialized kind of `Listener`. `Listeners` are related to changes in communication status. For each `Entity` type, one specific `Listener` is derived from this interface. In the following modules, the following `Listeners` are derived from this interface:

- `DomainParticipantListener`
- `ExtDomainParticipantListener`
- `TopicListener`
- `ExtTopicListener`
- `PublisherListener`
- `DataWriterListener`
- `SubscriberListener`
- `DataReaderListener`

The `Entity` type specific `Listener` interfaces are part of the application which must implement the interface operations. A user-defined class for these operations must be provided by the application which must extend from the specific Listener class. All Listener operations must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

PRISMTECH

Figure 11 DCPS Listeners

The base class `Listener` does not contain any operations.

3.1.5 Class Status

Each concrete `Entity` class has a set of `Status` attributes and for each attribute the `Entity` class provides an operation to read the value. Changes to `Status` attributes will affect associated `StatusCondition` and (invoked and associated) `Listener` objects.

The communication statuses whose changes can be communicated to the application depend on the `Entity`. The following table shows the relevant statuses for each `Entity`.

Table 13 Status Description Per Entity

Entity	Status Name	Meaning
Topic	INCONSISTENT_TOPIC_STATUS	Another `Topic` exists with the same name but with different characteristics.
	ALL_DATA_ DISPOSED_TOPIC_STATUS	All instances of the `Topic` have been disposed by the `dispose_all_data` operation on that topic.
Subscriber	DATA_ON_READERS_STATUS	New information is available.
DataReader	SAMPLE_REJECTED_STATUS	A (received) sample has been rejected.
	LIVELINESS_CHANGED_STATUS	The liveliness of one or more `DataWriter` objects that were writing instances read through the `DataReader` has changed. Some `DataWriter` have become "alive" or "not alive".
	REQUESTED_ DEADLINE_MISSED_STATUS	The deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected for a specific instance.
	REQUESTED_ INCOMPATIBLE_QOS_STATUS	A `QosPolicy` setting was incompatible with what is offered.
	DATA_AVAILABLE_STATUS	New information is available.
	SAMPLE_LOST_STATUS	A sample has been lost (never received).
	SUBSCRIPTION_MATCH_STATUS	The `DataReader` has found a `DataWriter` that matches the `Topic` and has compatible QoS.
DataWriter	LIVELINESS_LOST_STATUS	The liveliness that the `DataWriter` has committed through its `LivelinessQosPolicy` was not respected; thus `DataReader` objects will consider the `DataWriter` as no longer "alive".
	OFFERED_ DEADLINE_MISSED_STATUS	The deadline that the `DataWriter` has committed through its `DeadlineQosPolicy` was not respected for a specific instance.
	OFFERED_ INCOMPATIBLE_QOS_STATUS	A `QosPolicy` setting was incompatible with what was requested.
	PUBLICATION_MATCH_STATUS	The `DataWriter` has found `DataReader` that matches the `Topic` and has compatible QoS.

PRISMTECH

A `Status` attribute can be retrieved with the operation `get_<status_name>_status`. For example, to get the `InconsistentTopicStatus` value, the application must call the operation `get_inconsistent_topic_status`.

Conceptually associated with each `Entity` communication status is a logical `StatusChangedFlag`. This flag indicates whether that particular communication status has changed. The `StatusChangedFlag` is only conceptual, therefore, it is not important whether this flag actually exists.

For the plain communication `Status`, the `StatusChangedFlag` is initially set to `false`. It becomes `true` whenever the plain communication `Status` changes and it is reset to `false` each time the application accesses the plain communication `Status` via the proper `get_<status_name>_status` operation on the `Entity`.

A flag set means that a change has occurred since the last time the application has read its value.
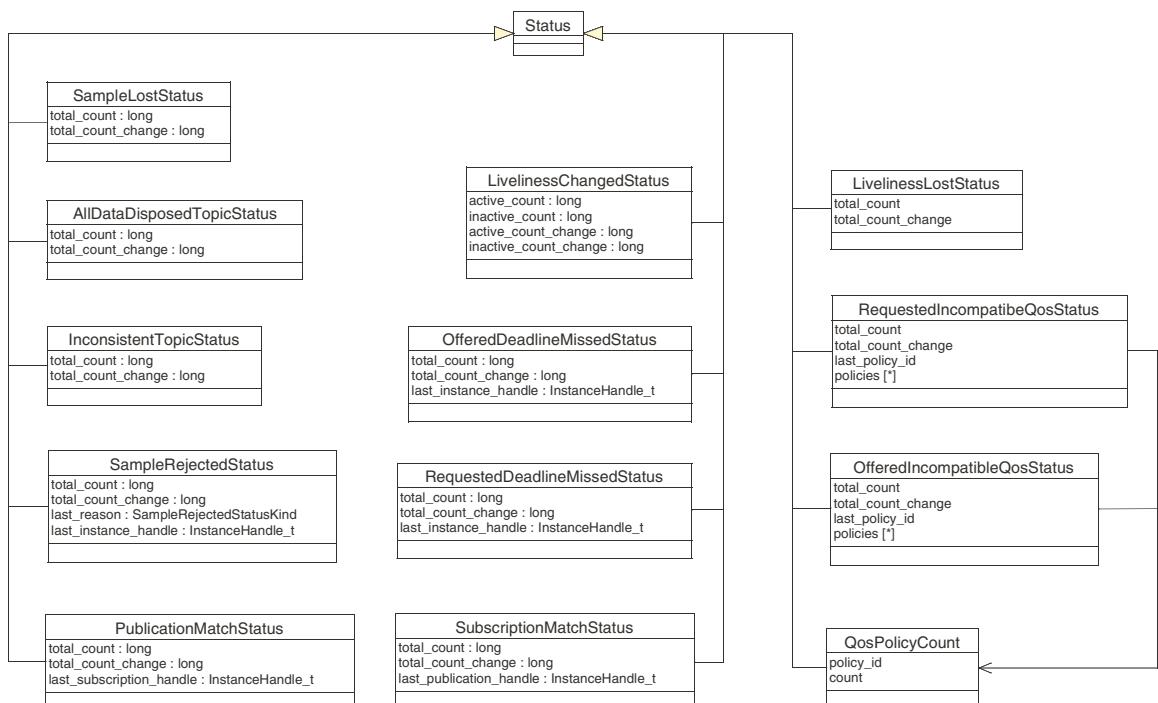
Figure 12 DCPS Status Values

Each `Status` attribute is implemented as a class without any operations. The interface description of these class is as follows:

```
// public class <name>Status
```

```
//
    public final class InconsistentTopicStatus
    {
        public int total_count;
        public int total_count_change;
    };
    public final class AllDataDisposedTopicStatus
    {
        public int total_count;
        public int total_count_change;
    };
    public final class SampleLostStatus
    {
        public int total_count;
        public int total_count_change;
    };
    public final class SampleRejectedStatusKind
    {
        public static final SampleRejectedStatusKind
                        NOT_REJECTED;
        public static final SampleRejectedStatusKind
                        REJECTED_BY_INSTANCES_LIMIT;
        public static final SampleRejectedStatusKind
                        REJECTED_BY_SAMPLES_LIMIT;
        public static final SampleRejectedStatusKind
                        REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT;
    };
    public final class SampleRejectedStatus
    {
        public int total_count;
        public int total_count_change;
        public SampleRejectedStatusKind last_reason;
        public long last_instance_handle;
    };
    public final class LivelinessLostStatus
    {
        public int total_count;
        public int total_count_change;
    };
    public final class LivelinessChangedStatus
    {
        public int alive_count;
        public int not_alive_count;
        public int alive_count_change;
        public int not_alive_count_change;
        public long last_publication_handle;};
    public final class OfferedDeadlineMissedStatus
    {
        public int total_count;
```

```
            public int total_count_change;
            public long last_instance_handle;
         };
         public final class RequestedDeadlineMissedStatus
         {
            public int total_count;
            public int total_count_change;
            public long last_instance_handle;
         };
         public final class OfferedIncompatibleQosStatus
         {
            public int total_count;
            public int total_count_change;
            public int last_policy_id;
            public QosPolicyCount policies[];
         };
         public final class RequestedIncompatibleQosStatus
         {
            public int total_count;
            public int total_count_change;
            public int last_policy_id;
            public QosPolicyCount policies[];
         };
         public final class PublicationMatchedStatus
         {
            public int total_count;
            public int total_count_change;
            public int current_count;
            public int current_count_change;
            public long last_subscription_handle;
         };
         public final class SubscriptionMatchedStatus
         {
            public int total_count;
            public int total_count_change;
            public int current_count;
            public int current_count_change;
            public long last_publication_handle;
         };
      //
      // implemented API operations
      //       <no operations>
      //
```

The following paragraphs describe the usage of each `<name>Status` struct.

3.1.5.1 **InconsistentTopicStatus**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class InconsistentTopicStatus
{
    public int total_count;
    public int total_count_change;
};
```

Description

This class contains the statistics about attempts to create other `Topics` with the same name but with different characteristics.

Attributes

int total_count - the total detected cumulative count of `Topic` creations, whose name matches the `Topic` to which this `Status` is attached and whose characteristics are inconsistent.

int total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

Detailed Description

This class contains the statistics about attempts to create other `Topics` with the same name but with different characteristics.

The attribute `total_count` holds the total detected cumulative count of `Topic` creations, whose name matches the `Topic` to which this `Status` is attached and whose characteristics are inconsistent.

The attribute `total_count_change` holds the incremental number of inconsistent `Topics`, since the last time the `Listener` was called or the `Status` was read.

3.1.5.2 **LivelinessChangedStatus**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class LivelinessChangedStatus
{
```

```
        public int alive_count;
        public int not_alive_count;
        public int alive_count_change;
        public int not_alive_count_change;
        public long last_publication_handle;
    };
```

Description

This class contains the statistics about whether the liveliness of one or more connected `DataWriter` objects has changed.

Attributes

int alive_count - the total count of currently alive `DataWriter` objects that write the topic read by the `DataReader` to which this `Status` is attached.

int not_alive_count - the total count of currently not alive `DataWriter` objects that wrote the topic read by the `DataReader` to which this `Status` is attached.

int alive_count_change - the change in `alive_count` since the last time the `Listener` was called or the `Status` was read.

int not_alive_count_change - the change in `not_alive_count` since the last time the `Listener` was called or the `Status` was read.

long last_publication_handle - handle to the last `DataWriter` whose change in liveliness caused this status to change.

Detailed Description

This class contains the statistics about whether the liveliness of one or more connected `DataWriter` objects that were writing instances read through the `DataReader` has changed. In other words, some `DataWriter` have become "alive" or "not alive".

The attribute `alive_count` holds the total number of currently alive `DataWriter` objects that write the topic read by the `DataReader` to which this `Status` is attached. This count increases when a newly matched `DataWriter` asserts its liveliness for the first time or when a `DataWriter` previously considered to be not alive reasserts its liveliness. The count decreases when a `DataWriter` considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason.

The attribute `not_alive_count` holds the total count of currently not alive `DataWriters` that wrote the topic read by the `DataReader` to which this `Status` is attached, and that are no longer asserting their liveliness. This count increases when a `DataWriter` considered alive fails to assert its liveliness and becomes not

alive for some reason other than the normal deletion of that `DataWriter`. It decreases when a previously not alive `DataWriter` either reasserts its liveliness or is deleted normally.

The attribute `alive_count_change` holds the change in `alive_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `not_alive_count_change` holds the change in `not_alive_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_publication_handle` contains the instance handle to the *`PublicationBuiltinTopicData`* instance that represents the last datawriter whose change in liveliness caused this status to change. Be aware that this handle belongs to *another* datareader, the *`PublicationBuiltinTopicData`*`DataReader` in the builtin-subscriber, and has no meaning in the context of the datareader from which the `LivelinessChangedStatus` was obtained. If the builtin-subscriber has not explicitly been obtained using `get_builtin_subscriber` on the `DomainParticipant`, then there is no *`PublicationBuiltinTopicData`*`DataReader` as well, in which case the `last_publication_handle` will be set to `HANDLE_NIL`.

3.1.5.3 LivelinessLostStatus

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class LivelinessLostStatus
{
    public int total_count;
    public int total_count_change;
};
```

Description

This class contains the statistics about whether the liveliness of the `DataWriter` to which this `Status` is attached has been committed through its `LivelinessQosPolicy`.

Attributes

`int total_count` - the total cumulative count of times the `DataWriter` to which this `Status` is attached failed to actively signal its liveliness within the offered liveliness period.

int total_count_change - the change in `total_count` since the last time the
`Listener` was called or the `Status` was read.

Detailed Description

This class contains the statistics about whether the liveliness of the `DataWriter` to
which this `Status` is attached has been committed through its
`LivelinessQosPolicy`. In other words, whether the `DataWriter` failed to
actively signal its liveliness within the offered liveliness period. In such a case, the
connected `DataReader` objects will consider the `DataWriter` as no longer
"alive".

The attribute `total_count` holds the total cumulative number of times that the
previously-alive `DataWriter` became not alive due to a failure to actively signal its
liveliness within its offered liveliness period. This count does not change when an
already not alive `DataWriter` simply remains not alive for another liveliness
period.

The attribute `total_count_change` holds the change in `total_count` since the
last time the `Listener` was called or the `Status` was read.

3.1.5.4 OfferedDeadlineMissedStatus

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
   public final class OfferedDeadlineMissedStatus
   {
       public int total_count;
       public int total_count_change;
       public long last_instance_handle;
   };
```

Description

This class contains the statistics about whether the deadline that the `DataWriter` to
which this `Status` is attached has committed through its `DeadlineQosPolicy`
was not respected for a specific instance.

Attributes

int total_count - the total cumulative count of times the `DataWriter` to which
this `Status` is attached failed to write within its offered deadline.

int total_count_change - the change in `total_count` since the last time the
`Listener` was called or the `Status` was read.

> `long last_instance_handle` - the handle to the last instance in the `DataWriter` to which this `Status` is attached, for which an offered deadline was missed.

Detailed Description

This class contains the statistics about whether the deadline that the `DataWriter` to which this `Status` is attached has committed through its `DeadlineQosPolicy` was not respected for a specific instance.

The attribute `total_count` holds the total cumulative number of offered deadline periods elapsed during which the `DataWriter` to which this `Status` is attached failed to provide data. Missed deadlines accumulate; that is, each deadline period the `total_count` will be incremented by one.

The attribute `total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_instance_handle` holds the handle to the last instance in the `DataWriter` to which this `Status` is attached, for which an offered deadline was missed.

3.1.5.5 OfferedIncompatibleQosStatus

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class OfferedIncompatibleQosStatus
{
    public int total_count;
    public int total_count_change;
    public int last_policy_id;
    public QosPolicyCount policies[];
};
```

Description

This class contains the statistics about whether an offered `QosPolicy` setting was incompatible with the requested `QosPolicy` setting.

Attributes

> `int total_count` - the total cumulative count of `DataReader` objects discovered by the `DataWriter` with the same `Topic` and `Partition` and with a requested `DataReaderQos` that was incompatible with the one offered by the `DataWriter`.

int total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

int last_policy_id - the id of one of the `QosPolicy` settings that was found to be incompatible with what was offered, the last time an incompatibility was detected.

QosPolicyCount policies[] - a list containing for each `QosPolicy` the total number of times that the concerned `DataWriter` discovered a `DataReader` for the same `Topic` and a requested `DataReaderQos` that is incompatible with the one offered by the `DataWriter`.

Detailed Description

This class contains the statistics about whether an offered `QosPolicy` setting was incompatible with the requested `QosPolicy` setting.

The Request/Offering mechanism is applicable between:

- the `DataWriter` and the `DataReader`. If the `QosPolicy` settings between `DataWriter` and `DataReader` are incompatible, no communication between them is established. In addition the `DataWriter` will be informed via a `REQUESTED_INCOMPATIBLE_QOS` status change and the `DataReader` will be informed via an `OFFERED_INCOMPATIBLE_QOS` status change.

- the `DataWriter` and the Durability Service (as a built-in `DataReader`). If the `QosPolicy` settings between `DataWriter` and the Durability Service are inconsistent, no communication between them is established. In that case data published by the `DataWriter` will not be maintained by the service and as a consequence will not be available for late joining `DataReaders`. The `QosPolicy` of the Durability Service in the role of `DataReader` is specified by the `DurabilityServiceQosPolicy` in the `Topic`.

- the Durability Service (as a built-in `DataWriter`) and the `DataReader`. If the `QosPolicy` settings between the Durability Service and the `DataReader` are inconsistent, no communication between them is established. In that case the Durability Service will not publish historical data to late joining `DataReaders`. The `QosPolicy` of the Durability Service in the role of `DataWriter` is specified by the `DurabilityServiceQosPolicy` in the `Topic`.

The attribute `total_count` holds the total cumulative count of `DataReader` objects discovered by the `DataWriter` with the same `Topic` and a requested `DataReaderQos` that was incompatible with the one offered by the `DataWriter`.

The attribute `total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_policy_id` holds the id of one of the `QosPolicy` settings that was found to be incompatible with what was offered, the last time an incompatibility was detected.

The attribute `policies` holds a list containing for each `QosPolicy` the total number of times that the concerned `DataWriter` discovered an incompatible `DataReader` for the same `Topic`. Each element in the list represents a counter for a different `QosPolicy`, identified by a corresponding unique index number. A named list of all index numbers is expressed as a set of constants in the API. See *Table 13* for an overview of all these constants.

Table 14 Overview of All Named QosPolicy Indexes

Index name	Index Value
`INVALID_QOS_POLICY_ID`	0
`USERDATA_QOS_POLICY_ID`	1
`DURABILITY_QOS_POLICY_ID`	2
`PRESENTATION_QOS_POLICY_ID`	3
`DEADLINE_QOS_POLICY_ID`	4
`LATENCYBUDGET_QOS_POLICY_ID`	5
`OWNERSHIP_QOS_POLICY_ID`	6
`OWNERSHIPSTRENGTH_QOS_POLICY_ID`	7
`LIVELINESS_QOS_POLICY_ID`	8
`TIMEBASEDFILTER_QOS_POLICY_ID`	9
`PARTITION_QOS_POLICY_ID`	10
`RELIABILITY_QOS_POLICY_ID`	11
`DESTINATIONORDER_QOS_POLICY_ID`	12
`HISTORY_QOS_POLICY_ID`	13
`RESOURCELIMITS_QOS_POLICY_ID`	14
`ENTITYFACTORY_QOS_POLICY_ID`	15
`WRITERDATALIFECYCLE_QOS_POLICY_ID`	16
`READERDATALIFECYCLE_QOS_POLICY_ID`	17
`TOPICDATA_QOS_POLICY_ID`	18
`GROUPDATA_QOS_POLICY_ID`	19
`TRANSPORTPRIORITY_QOS_POLICY_ID`	20
`LIFESPAN_QOS_POLICY_ID`	21
`DURABILITYSERVICE_QOS_POLICY_ID`	22

PRISMTECH

3.1.5.6 **PublicationMatchedStatus**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class PublicationMatchedStatus
{
  public int total_count;
  public int total_count_change;
  public int current_count;
  public int current_count_change;
  public long last_subscription_handle;
};
```

This class contains the statistics about the discovered number of matching DataReaders currently connected to the owner of this status, and of the cumulative number of DataReaders that has connected to the owner of this status over time.

Attributes

int total_count - Total cumulative count of DataReaders compatible with the concerned DataWriter.

int total_count_change - The change in total_count since the last time the Status was read.

int current_count - Total count of DataReaders that are currently available and compatible with the DataWriter.

int current_count_change - The change in current_count since the last time the Status was read.

long last_subscription_handle - Handle to the last DataReader that matched the DataWriter causing the status to change.

Detailed Description

This class contains the statistics about the discovered number of DataReaders that are compatible with the DataWriter to which the Status is attached. DataReader and DataWriter are compatible if they use the same Topic and if the QoS requested by the DataReader is compatible with that offered by the DataWriter. A DataReader will automatically connect to a matching DataWriter, but will disconnect when that DataReader is deleted, when either changes its QoS into an incompatible value, or when either puts its matching counterpart on its ignore-list using the ignore_subscription or ignore_publication operations on the DomainParticipant.

The `total_count` includes DataReaders that have already been disconnected, while in the `current_count` only the currently connected DataReaders are considered.

3.1.5.7 **RequestedDeadlineMissedStatus**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class RequestedDeadlineMissedStatus
{
    public int total_count;
    public int total_count_change;
    public long last_instance_handle;
};
```

Description

This class contains the statistics about whether the deadline that the `DataReader` to which this `Status` is attached was expecting through its `DeadlineQosPolicy` was not respected for a specific instance.

Attributes

int total_count - the total cumulative count of the missed deadlines detected for any instance read by the `DataReader` to which this `Status` is attached.

int total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

long last_instance_handle - the handle to the last instance in the `DataReader` to which this `Status` is attached for which a missed deadline was detected.

Detailed Description

This class the statistics about whether the deadline that the `DataReader` to which this `Status` is attached was expecting through its `DeadlineQosPolicy` was not respected for a specific instance. Missed deadlines accumulate, that is, each deadline period the `total_count` will be incremented by one for each instance for which data was not received.

The attribute `total_count` holds the total cumulative count of the missed deadlines detected for any instance read by the `DataReader`.

The attribute `total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_instance_handle` holds the handle to the last instance in the `DataReader` for which a missed deadline was detected.

3.1.5.8 RequestedIncompatibleQosStatus

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class RequestedIncompatibleQosStatus
{
    public int total_count;
    public int total_count_change;
    public int last_policy_id;
    public QosPolicyCount policies[];
};
```

Description

This class contains the statistics about whether a requested `QosPolicy` setting was incompatible with the offered `QosPolicy` setting.

Attributes

int total_count - the total cumulative count of `DataWriter` objects, discovered by the `DataReader` to which this `Status` is attached, with the same `Topic` and an offered `DataWriterQos` that was incompatible with the one requested by the `DataReader`.

int total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

int last_policy_id - the `<name>_QOS_POLICY_ID` of one of the `QosPolicies` that was found to be incompatible with what was requested, the last time an incompatibility was detected.

QosPolicyCount policies[] - a list containing (for each `QosPolicy`) the total number of times that the concerned `DataReader` discovered a `DataWriter` with the same `Topic` and an offered `DataWriterQos` that is incompatible with the one requested by the `DataReader`.

Detailed Description

This class contains the statistics about whether a requested `QosPolicy` setting was incompatible with the offered `QosPolicy` setting.

The Request/Offering mechanism is applicable between:

- the `DataWriter` and the `DataReader`. If the `QosPolicy` settings between `DataWriter` and `DataReader` are incompatible, no communication between them is established. In addition the `DataWriter` will be informed via a `REQUESTED_INCOMPATIBLE_QOS` status change and the `DataReader` will be informed via an `OFFERED_INCOMPATIBLE_QOS` status change.

- the `DataWriter` and the Durability Service (as a built-in `DataReader`). If the `QosPolicy` settings between `DataWriter` and the Durability Service are inconsistent, no communication between them is established. In that case data published by the `DataWriter` will not be maintained by the service and as a consequence will not be available for late joining `DataReaders`. The `QosPolicy` of the Durability Service in the role of `DataReader` is specified by the `DurabilityServiceQosPolicy` in the `Topic`.

- the Durability Service (as a built-in `DataWriter`) and the `DataReader`. If the `QosPolicy` settings between the Durability Service and the `DataReader` are inconsistent, no communication between them is established. In that case the Durability Service will not publish historical data to late joining `DataReaders`. The `QosPolicy` of the Durability Service in the role of `DataWriter` is specified by the `DurabilityServiceQosPolicy` in the `Topic`.

The attribute `total_count` holds the total cumulative count of `DataWriter` objects discovered by the `DataReader` with the same `Topic` and an offered `DataWriterQos` that was incompatible with the one requested by the `DataReader`.

The attribute `total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_policy_id` holds the `<name>_QOS_POLICY_ID` of one of the `QosPolicies` that was found to be incompatible with what was requested, the last time an incompatibility was detected.

The attribute `policies` holds a list containing for each `QosPolicy` the total number of times that the concerned `DataReader` discovered an incompatible `DataWriter` for the same `Topic`. Each element in the list represents a counter for a different `QosPolicy`, identified by a corresponding unique index number. A named list of all index numbers is expressed as a set of constants in the API. See Table 14, *Overview of All Named QosPolicy Indexes*, on page 106 for an overview of all these constants.

3.1.5.9 **SampleLostStatus**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class SampleLostStatus
{
    public int total_count;
    public int total_count_change;
};
```

Description

This class contains the statistics about whether a sample has been lost (never received).

Attributes

int total_count - the total cumulative count of all samples lost across all instances of data published under the `Topic`.

int total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

Detailed Description

This class contains the statistics about whether a sample has been lost (never received). The status is independent of the differences in instances, in other words, it includes all samples lost across all instances of data published under the `Topic`.

`total_count` holds the total cumulative count of all samples lost across all instances of data published under the `Topic`.

`total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

3.1.5.10 **SampleRejectedStatus**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class SampleRejectedStatusKind
        {
            public static final SampleRejectedStatusKind
                    NOT_REJECTED;
            public static final SampleRejectedStatusKind
                    REJECTED_BY_INSTANCES_LIMIT;
            public static final SampleRejectedStatusKind
                    REJECTED_BY_SAMPLES_LIMIT;
            public static final SampleRejectedStatusKind
```

```
                          REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT;
        }
        public final class SampleRejectedStatus
        {
            public int total_count;
            public int total_count_change;
            public SampleRejectedStatusKind last_reason;
            public long last_instance_handle;
    };
```

Description

This class contains the statistics about samples that have been rejected.

Attributes

int total_count - the total cumulative count of samples rejected by the
    DataReader to which this Status is attached.

int total_count_change - the change in total_count since the last time the
    Listener was called or the Status was read.

SampleRejectedStatusKind last_reason - the reason for rejecting the last
    sample.

long last_instance_handle - the handle to the instance which would have
    been updated by the last sample that was rejected.

Detailed Description

This class contains the statistics about whether a received sample has been rejected.

The attribute total_count holds the total cumulative count of samples rejected by
the DataReader to which this Status is attached.

The attribute total_count_change holds the change in total_count since the
last time the Listener was called or the Status was read.

The attribute last_reason holds the reason for rejecting the last sample. The
attribute can have the following values:

- *NOT_REJECTED* - no sample has been rejected yet.

- *REJECTED_BY_INSTANCES_LIMIT* - the sample was rejected because it would
  exceed   the   maximum   number   of   instances   set   by   the
  ResourceLimitsQosPolicy.

- *REJECTED_BY_SAMPLES_LIMIT* - the sample was rejected because it would
  exceed the maximum number of samples set by the ResourceLimits
  QosPolicy.

- *REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT* - the sample was rejected because it would exceed the maximum number of samples per instance set by the `ResourceLimitsQosPolicy`.

The attribute `last_instance_handle` holds the handle to the instance which would have updated by the last sample that was rejected.

3.1.5.11 SubscriptionMatchedStatus

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public final class SubscriptionMatchedStatus
{
    public int total_count;
    public int total_count_change;
    public int current_count;
    public int current_count_change;
    public long last_publication_handle;
};
```

This class contains the statistics about the discovered number of matching DataWriters currently connected to the owner of this status, and of the cumulative number of DataWriters that has connected to the owner of this status over time.

Attributes

int total_count - Total cumulative count of DataWriters compatible with the concerned DataReader.

int total_count_change - The change in `total_count` since the last time the Status was read.

int current_count - Total count of DataWriters that are currently available and compatible with the DataWriter.

int current_count_change - The change in `current_count` since the last time the Status was read.

long last_publication_handle - Handle to the last DataWriter that matched the DataReader causing the status to change.

Detailed Description

This class contains the statistics about the discovered number of DataWriters that are compatible with the DataReader to which the Status is attached. DataWriter and DataReader are compatible if they use the same Topic and if the QoS requested by the DataReader is compatible with that offered by the DataWriter. A DataWriter will

automatically connect to a matching DataReader, but will disconnect when that DataWriter is deleted, when either changes its QoS into an incompatible value, or when either puts its matching counterpart on its ignore-list using the `ignore_subscription` or `ignore_publication` operations on the DomainParticipant.

The `total_count` includes DataWriters that have already been disconnected, while in the `current_count` only the currently connected DataWriters are considered.

3.1.5.12 AllDataDisposedTopicStatus

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public final class AllDataDisposedTopicStatus
{   int total_count;
     int total_count_change;  }
```

Description

This class contains the statistics about the occurence of the `ALL_DATA_DISPOSED_TOPIC_STATUS` event on the Topic to which this Status is attached.

Attributes

int total_count - the total detected cumulative count of `ALL_DATA_DISPOSED_TOPIC_STATUS` events.

int total_count_change - the change in `total_count` since the last time the Status was read.

Detailed Description

This class contains the statistics about the occurence of the `ALL_DATA_DISPOSED_TOPIC_STATUS` event on the Topic to which this Status is attached. The Status is directly related to the invocation of the `DDS.Topic.dispose_all_data()` operation. Statistics are only kept when all instances are disposed using this operation, not when instances are disposed seperately by individual dispose calls.

3.1.6 Class WaitSet

A `WaitSet` object allows an application to wait until one or more of the attached `Condition` objects evaluates to `true` or until the timeout expires.

The `WaitSet` has no factory and must be created by the application. It is directly created as an object by using `WaitSet` constructors.
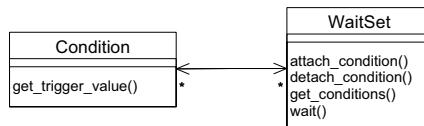
Figure 13 DCPS WaitSets

The interface description of this class is as follows:

```
public class WaitSet
{
//
// implemented API operations
//
    public int
        _wait
            (ConditionSeqHolder active_conditions,
            Duration_t timeout);
    public int
        attach_condition
            (Condition cond);
    public int
        detach_condition
            (Condition cond);
    public int
        get_conditions
            (ConditionSeqHolder attached_conditions);
};
```

The following paragraphs describe the usage of all WaitSet operations.

3.1.6.1 attach_condition

Scope

```
DDS.WaitSet
```

Synopsis

```
import DDS.*;
public int
    attach_condition
        (Condition cond);
```

Description

This operation attaches a `Condition` to the `WaitSet`.

Parameters

in Condition cond - a reference to a `Condition`.

Return Value

int - Possible return codes of the operation are:
> `RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation attaches a `Condition` to the `WaitSet`. The parameter `cond` must be either a `ReadCondition, QueryCondition, StatusCondition` or `GuardCondition`. To get this parameter see:

- `ReadCondition` created by `create_readcondition`
- `QueryCondition` created by `create_querycondition`
- `StatusCondition` retrieved by `get_statuscondition` on an `Entity`
- `GuardCondition` created by the Java operation `new`.

When a `GuardCondition` is initially created, the `trigger_value` is `false`.

When a `Condition`, whose `trigger_value` evaluates to `true`, is attached to a `WaitSet` that is currently being waited on (using the `wait` operation), the `WaitSet` will unblock immediately.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Condition` is attached to the `WaitSet`
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - the parameter `cond` is not a valid `Condition` reference.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.1.6.2 detach_condition

Scope

```
DDS.WaitSet
```

Synopsis

```
import DDS.*;
public int
    detach_condition
```

```
(Condition cond);
```

Description

This operation detaches a `Condition` from the `WaitSet`.

Parameters

in Condition cond - a reference to a `Condition` in the `WaitSet`.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`,
   `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation detaches a `Condition` from the `WaitSet`. If the `Condition` was not attached to this `WaitSet`, the operation returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Condition` is detached from the `WaitSet`.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter `cond` is not a valid `Condition` reference.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- RETCODE_PRECONDITION_NOT_MET - the `Condition` was not attached to this `WaitSet`.

3.1.6.3 get_conditions

Scope

```
DDS.WaitSet
```

Synopsis

```
import DDS.*;
public int
   get_conditions
      (ConditionSeqHolder attached_conditions);
```

Description

This operation retrieves the list of attached conditions.

Parameters

inout ConditionSeqHolder attached_conditions - a Holder to a sequence which is used to pass the list of attached conditions.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation retrieves the list of attached conditions in the WaitSet. The parameter attached_conditions is a Holder to a sequence which afterwards will refer to the sequence of attached conditions. The array inside the attached_conditions Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the get_conditions operation. If the pre-allocated array is not big enough to hold the the number of attached Conditions, the sequence will automatically be (re-)allocated to fit the required size. The resulting sequence will either be an empty sequence, meaning there were no conditions attached, or will contain a list of ReadCondition, QueryCondition, StatusCondition and GuardCondition. These conditions previously have been attached by attach_condition and were created by there respective create operation:

- ReadCondition created by create_readcondition
- QueryCondition created by create_querycondition
- StatusCondition retrieved by get_statuscondition on an Entity
- GuardCondition created by the Java operation new.

Return Code

When the operation returns:

- *RETCODE_OK* - the list of attached conditions is returned
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.1.6.4 **wait**

Scope

```
DDS.WaitSet
```

PRISMTECH

Synopsis

```
import DDS.*;
public int
   _wait
      (ConditionSeqHolder active_conditions,
       Duration_t timeout);
```

Description

This operation allows an application thread to wait for the occurrence of at least one of the conditions that is attached to the `WaitSet`.

Parameters

inout ConditionSeqHolder active_conditions - parameter `active_conditions` is a Holder to a sequence, which is used to pass the list of all the attached conditions that have a `trigger_value` of `true`.

in Duration_t timeout - the maximum duration to block for the `wait`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_TIMEOUT` or `RETCODE_ PRECONDITION_NOT_MET`.

Detailed Description

This operation allows an application thread to wait for the occurrence of at least one of the `conditions` to evaluate to `true` that is attached to the `WaitSet`. If all of the conditions attached to the `WaitSet` have a `trigger_value` of `false`, the `wait` operation will block the calling thread. The result of the operation is the continuation of the application thread after which the result is left in `active_conditions`. This is a Holder for a sequence, which will contain the list of all the attached conditions that have a `trigger_value` of `true`. The array inside the `active_conditions` Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the `_wait` operation. If the pre-allocated array is not big enough to hold the the number of triggered `Conditions`, the array will automatically be (re-)allocated to fit the required size. The parameter `timeout` specifies the maximum duration for the `wait` to block the calling application thread (when none of the attached `conditions` have a trigger value of `true`). In that case the return value is `RETCODE_TIMEOUT` and the `active_conditions` sequence is left empty. Since it is not allowed for more than one application thread to be waiting on the same `WaitSet`, the operation returns

immediately with the value RETCODE_PRECONDITION_NOT_MET when the _wait operation is invoked on a WaitSet which already has an application thread blocking on it.

Return Code

When the operation returns:

- *RETCODE_OK* - at least one of the attached conditions has a trigger_value of true.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_TIMEOUT* - the timeout has elapsed without any of the attached conditions becoming true.

- *RETCODE_PRECONDITION_NOT_MET* - the WaitSet already has an application thread blocking on it.

3.1.7 Class Condition

This class is the base class for all the conditions that may be attached to a WaitSet. This base class is specialized in three classes by the Data Distribution Service: GuardCondition, StatusCondition and ReadCondition (also there is a QueryCondition which is a specialized ReadCondition).

Each Condition has a trigger_value that can be true or false and is set by the Data Distribution Service (except a GuardCondition) depending on the evaluation of the Condition.
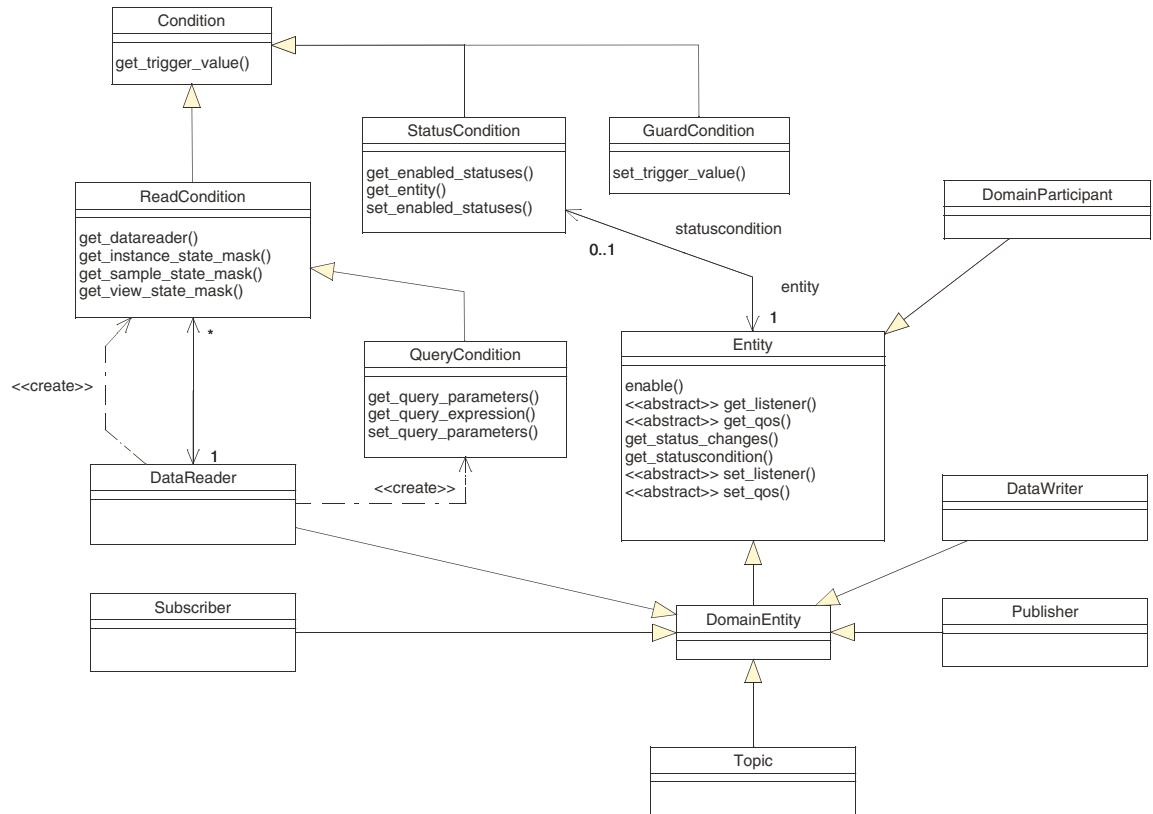
Figure 14 DCPS Conditions

The interface description is as follows:

```
public interface Condition
{
//
// implemented API operations
//
    public boolean
       get_trigger_value
           (void);
};
```

The next paragraph describes the usage of the `Condition` operation.

3.1.7.1 **get_trigger_value**

Scope

```
DDS.Condition
```

Synopsis

```
import DDS.*;
public boolean
    get_trigger_value
        (void);
```

Description

This operation returns the `trigger_value` of the Condition.

Parameters

<none>

Return Value

boolean - is the `trigger_value`.

Detailed Description

A `Condition` has a `trigger_value` that can be `true` or `false` and is set by the Data Distribution Service (except a `GuardCondition`). This operation returns the `trigger_value` of the `Condition`.

3.1.8 Class GuardCondition

A `GuardCondition` object is a specific `Condition` whose `trigger_value` is completely under the control of the application. The `GuardCondition` has no factory and must be created by the application. The `GuardCondition` is directly created as an object by using the `GuardCondition` constructor. When a `GuardCondition` is initially created, the `trigger_value` is `false`. The purpose of the `GuardCondition` is to provide the means for an application to manually wake up a `WaitSet`. This is accomplished by attaching the `GuardCondition` to the `Waitset` and setting the `trigger_value` by means of the `set_trigger_value` operation.

The interface description of this class is as follows:

```
public interface GuardCondition
{
//
// extends interface Condition
//
// public boolean
//     get_trigger_value
//         (void);
//
// implemented API operations
//
    public int
```

PRISMTECH

```
          set_trigger_value
             (boolean value);
    };
```

The following paragraphs describe the usage of all `GuardCondition` operations. The inherited operation is listed but not fully described since it is not implemented in this class. The full description of this operation is given in the class from which it is inherited. This is described in their respective paragraph.

3.1.8.1 **get_trigger_value (inherited)**

This operation is inherited and therefore not described here. See the class `Condition` for further explanation.

Synopsis

```
import DDS.*;
public boolean
   get_trigger_value
      (void);
```

3.1.8.2 **set_trigger_value**

Scope

```
DDS.GuardCondition
```

Synopsis

```
import DDS.*;
public int
   set_trigger_value
      (boolean value);
```

Description

This operation sets the `trigger_value` of the `GuardCondition`.

Parameters

in boolean value - the boolean value to which the `GuardCondition` is set.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK` or `RETCODE_ERROR`.

Detailed Description

A `GuardCondition` object is a specific `Condition` which `trigger_value` is completely under the control of the application. This operation must be used by the application to manually wake-up a `WaitSet`. This operation sets the

trigger_value of the GuardCondition to the parameter value. The GuardCondition is directly created using the GuardCondition constructor. When a GuardCondition is initially created, the trigger_value is false.

Return Code

When the operation returns:

- *RETCODE_OK* - the specified trigger_value has successfully been applied.
- *RETCODE_ERROR* - an internal error has occurred.

3.1.9 Class StatusCondition

Entity objects that have status attributes also have a StatusCondition, access is provided to the application by the get_statuscondition operation.

The communication statuses whose changes can be communicated to the application depend on the Entity. The following table shows the relevant statuses for each Entity.

Table 15 Status per Entity

Entity	Status Name
Topic	INCONSISTENT_TOPIC_STATUS
	ALL_DATA_DISPOSED_TOPIC_STATUS
Subscriber	DATA_ON_READERS_STATUS
DataReader	SAMPLE_REJECTED_STATUS
	LIVELINESS_CHANGED_STATUS
	REQUESTED_DEADLINE_MISSED_STATUS
	REQUESTED_INCOMPATIBLE_QOS_STATUS
	DATA_AVAILABLE_STATUS
	SAMPLE_LOST_STATUS
	SUBSCRIPTION_MATCH_STATUS
DataWriter	LIVELINESS_LOST_STATUS
	OFFERED_DEADLINE_MISSED_STATUS
	OFFERED_INCOMPATIBLE_QOS_STATUS
	PUBLICATION_MATCH_STATUS

The trigger_value of the StatusCondition depends on the communication statuses of that Entity (e.g., missed deadline) and also depends on the value of the StatusCondition attribute mask (enabled_statuses mask). A StatusCondition can be attached to a WaitSet in order to allow an application to suspend until the trigger_value has become true.

PRISMTECH

The `trigger_value` of a `StatusCondition` will be `true` if one of the enabled `StatusChangedFlags` is set. That is, `trigger_value==false` only if all the values of the `StatusChangedFlags` are `false`.

The sensitivity of the `StatusCondition` to a particular communication status is controlled by the list of `enabled_statuses` set on the condition by means of the `set_enabled_statuses` operation.

When the `enabled_statuses` are not changed by the `set_enabled_statuses` operation, all statuses are enabled by default.

The interface description of this class is as follows:

```
public interface StatusCondition
{
//
// extends interface Condition
//
// boolean
//    get_trigger_value
//        (void);
//
// implemented API operations
//
    public int
        get_enabled_statuses
            (void);

    public int
        set_enabled_statuses
            (int mask);

    public Entity
        get_entity
            (void);
};
```

The following paragraphs describe the usage of all `StatusCondition` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.1.9.1 get_enabled_statuses

Scope

```
DDS.StatusCondition
```

Synopsis

```
import DDS.*;
public int
```

```
get_enabled_statuses
    (void);
```

Description

This operation returns the list of enabled communication statuses of the `StatusCondition`.

Parameters

<none>

Return Value

int - Result is a bit mask in which each bit shows which status is taken into account for the `StatusCondition`.

Detailed Description

The `trigger_value` of the `StatusCondition` depends on the communication status of that `Entity` (e.g., missed deadline, loss of information, etc.), 'filtered' by the set of `enabled_statuses` on the `StatusCondition`.

This operation returns the list of communication statuses that are taken into account to determine the `trigger_value` of the `StatusCondition`. This operation returns the statuses that were explicitly set on the last call to `set_enabled_statuses` or, if `set_enabled_statuses` was never called, the default list.

The result value is a bit mask in which each bit shows which status is taken into account for the `StatusCondition`. The relevant bits represents one of the following statuses:

- `INCONSISTENT_TOPIC_STATUS`
- `ALL_DATA_DISPOSED_TOPIC_STATUS`
- `OFFERED_DEADLINE_MISSED_STATUS`
- `REQUESTED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_ON_READERS_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_LOST_STATUS`
- `LIVELINESS_CHANGED_STATUS`
- `PUBLICATION_MATCHED_STATUS`

PRISMTECH

- `SUBSCRIPTION_MATCHED_STATUS`

Each status bit is declared as a constant and can be used in an `AND` operation to check the status bit against the result of type `int`.

Not all statuses are relevant to all `Entity` objects. See the respective `Listener` objects for each `Entity` for more information.

3.1.9.2 **get_entity**

Scope

```
DDS.StatusCondition
```

Synopsis

```
import DDS.*;
public Entity
    get_entity
        (void);
```

Description

This operation returns the `Entity` associated with the `StatusCondition` or the `null` reference.

Parameters

<none>

Return Value

Entity - Result value is a reference to the `Entity` associated with the `StatusCondition` or the `null` reference.

Detailed Description

This operation returns the `Entity` associated with the `StatusCondition`. Note that there is exactly one `Entity` associated with each `StatusCondition`. When the `Entity` was already deleted (there is no associated `Entity` any more), the `null` reference is returned.

3.1.9.3 **get_trigger_value (inherited)**

This operation is inherited and therefore not described here. See the class `Condition` for further explanation.

Synopsis

```
import DDS.*;
public boolean
    get_trigger_value
```

```
                    (void);
```

3.1.9.4 set_enabled_statuses

Scope
```
DDS.StatusCondition
```

Synopsis
```
import DDS.*;
public int
    set_enabled_statuses
        (int mask);
```

Description

This operation sets the list of communication statuses that are taken into account to determine the `trigger_value` of the `StatusCondition`.

Parameters

in int mask - a bit mask in which each bit sets the status which is taken into account for the `StatusCondition`.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_ALREADY_DELETED`.

Detailed Description

The `trigger_value` of the `StatusCondition` depends on the communication status of that `Entity` (e.g., missed deadline, loss of information, etc.), 'filtered' by the set of `enabled_statuses` on the `StatusCondition`.

This operation sets the list of communication statuses that are taken into account to determine the `trigger_value` of the `StatusCondition`. This operation may change the `trigger_value` of the `StatusCondition`.

`WaitSet` objects behaviour depend on the changes of the `trigger_value` of their attached `Conditions`. Therefore, any `WaitSet` to which the `StatusCondition` is attached is potentially affected by this operation.

If this function is not invoked, the default list of `enabled_statuses` includes all the statuses.

The parameter `mask` is a bit mask in which each bit shows which status is taken into account for the `StatusCondition`. The relevant bits represents one of the following statuses:

- `INCONSISTENT_TOPIC_STATUS`

- ALL_DATA_DISPOSED_TOPIC_STATUS
- OFFERED_DEADLINE_MISSED_STATUS
- REQUESTED_DEADLINE_MISSED_STATUS
- OFFERED_INCOMPATIBLE_QOS_STATUS
- REQUESTED_INCOMPATIBLE_QOS_STATUS
- SAMPLE_LOST_STATUS
- SAMPLE_REJECTED_STATUS
- DATA_ON_READERS_STATUS
- DATA_AVAILABLE_STATUS
- LIVELINESS_LOST_STATUS
- LIVELINESS_CHANGED_STATUS
- PUBLICATION_MATCHED_STATUS
- SUBSCRIPTION_MATCHED_STATUS

Each status bit is declared as a constant and can be used in an OR operation to set the status bit in the parameter `mask` of type `int`. The constants are implemented as an interface.

Not all statuses are relevant to all `Entity` objects. See the respective `Listener` objects for each `Entity` for more information.

Return Code

When the operation returns:

- *RETCODE_OK* - the list of communication statuses is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `StatusCondition` has already been deleted.

3.1.10 **Class ErrorInfo**

The *ErrorInfo* mechanism is an OpenSplice-specific extension to the OMG-DDS standard, that can help DDS users to get a more finegrained overview of the context of an error. The DDS specification only mandates that functions return a `ReturnCode_t` value as a broad categorization of potential types of problems (there are 12 possible `ReturnCode_t` values, of which 11 indicate some kind of error), but factory operations do not even have this mechanism at their disposal since they return the object they were requested to create.

The *ErrorInfo* was added to OpenSplice for the following reasons:

- It can provide context for errors that occur in factory operations (*e.g.* when `create_topic` returns `null`).

- It can provide an `ErrorCode_t` value, that represents a much more fine-grained error categorization than the `ReturnCode_t` (21 categories *vs.* the 11 categories provided by `ReturnCode_t`).

- It can provide an error description that can give a much more dedicated explanation of the exact circumstances of the error.

- It can provide the name of the function call/component that caused the error.

- It can provide source code location where the error occured (file name + line number).

- It can provide a stacktrace of the thread that ran into the error.

The *ErrorInfo* class obtains its information from the API-level log messages recorded by the internal mechanisms of the data distribution service. These are messages that are, by default, also written to the `ospl-info.log` file. The application can access this information through an *ErrorInfo* object, and take appropriate action based on the contents of this information. The *ErrorInfo* has no factory and an instance of the class can be created by the application by calling its default (empty) constructor.

The interface of this class is as follows:

```
public class ErrorInfo
{
   public int
     update();

    public int
      get_code
      (DDS.ErrorCodeHolder code);

    public int
      get_code
      (org.OMG.CORBA.IntHolder code);

    public int
      get_message
      (DDS.StringHolder message);

    public int
      get_message
      (org.omg.CORBA.StringHolder message);

    public int
      get_location
      (DDS.StringHolder location);

    public int
      get_location
```

```
       (org.omg.CORBA.StringHolder location);

    public int
      get_source_line
      (DDS.StringHolder sourceLine);

    public int
      get_source_line
      (org.omg.CORBA.StringHolder sourceLine);

    public int
      get_stack_trace
      (DDS.StringHolder stackTrace);

    public int
      get_stack_trace
      (org.omg.CORBA.StringHolder stackTrace);
};
```

The following sections describe the usage of all *ErrorInfo* operations.

3.1.10.1 **update**

Scope

```
DDS.ErrorInfo
```

Synopsis

```
import DDS.*;
public int
    update();
```

Description

This operation updates the `ErrorInfo` object with the latest available information.

Parameters

```
<none>
```

Return Value

int - Possible return codes of the operation are:
    RETCODE_OK, RETCODE_NO_DATA.

Detailed Description

This operation requests the latest error information from the data distribution service and stores it in the `ErrorInfo` object. The error information remains available in the `ErrorInfo` object until a new error occurs *and* the update operation is

explicitly invoked on the `ErrorInfo` object. If the information is successfully updated, `RETCODE_OK` is returned. If no information is available because no error has occurred yet, `RETCODE_NO_DATA` is returned.

3.1.10.2 **get_code**

Scope

```
DDS.ErrorInfo
```

Synopsis

```
public int
  get_code
  (DDS.ErrorCodeHolder code);
```

  or

```
public int
  get_code
  (org.OMG.CORBA.IntHolder code);
```

Description

This operation retrieves the error code of the last error message.

Two variants of the operation exist; the first can be used under all circumstances. The second is purely meant as a convenience when using the API in cohabitation with CORBA.

NOTE: This operation is not consistently implemented everywhere: various kinds of errors are still categorized as 'UNDEFINED'.

Parameters

in `DDS.ErrorCodeHolder code` - The holder in which the error code will be stored.

    or

in `org.omg.CORBA.IntHolder code` - The holder in which the error code will be stored.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_NO_DATA`.

Detailed Description

This operation stores the error code of the latest error in the provided holder class. The stored value is of type `int` and the vlaues range from `0` to `21`.

Table 16 below contains a list of all supported error code values and their meaning.

PRISMTECH

Table 16 All `ErrorInfo` values

Label	Value	Meaning.
ERRORCODE_UNDEFINED	0	Error has not (yet) been categorized.
ERRORCODE_ERROR	1	Unexpected error.
ERRORCODE_OUT_OF_RESOURCES	2	Not enough resources to complete the operation.
ERRORCODE_CREATION_KERNEL_ENTITY_FAILED	3	The kernel was not able to create the entity. Probably there is not enough shared memory available.
ERRORCODE_INVALID_VALUE	4	A value is passed that is outside its valid bounds.
ERRORCODE_INVALID_DURATION	5	A Duration is passed that is outside its valid bounds or that has not been normalized properly.
ERRORCODE_INVALID_TIME	6	A Time is passed that is outside its valid bounds or that has not been normalized properly.
ERRORCODE_ENTITY_INUSE	7	Attempted to delete an entity that is still in use.
ERRORCODE_CONTAINS_ENTITIES	8	Attempted to delete a factory that still contains entities.
ERRORCODE_ENTITY_UNKNOWN	9	A pointer to an unknown entity has been passed.
ERRORCODE_HANDLE_NOT_REGISTERED	10	A handle has been passed that is no longer in use.
ERRORCODE_HANDLE_NOT_MATCH	11	A handle has been passed to an entity to which it does not belong.
ERRORCODE_HANDLE_INVALID	12	An unknown handle has been passed.
ERRORCODE_INVALID_SEQUENCE	13	A sequence has been passed that has inconsistent variables (*e.g.* length > maximum, buffer equals NULL while maximum > 0, *etc.*)
ERRORCODE_UNSUPPORTED_VALUE	14	A value has been passed that is not (yet) supported.
ERRORCODE_INCONSISTENT_VALUE	15	A value has been passed that is inconsistent
ERRORCODE_IMMUTABLE_QOS_POLICY	16	Attempted to modify a QosPolicy that is immutable.

<div align="center">

Table 16 All `ErrorInfo` values (continued)

</div>

Label	Value	Meaning.
`ERRORCODE_INCONSISTENT_QOS`	17	Attempted to set QosPolicy values that are mutually inconsistent.
`ERRORCODE_UNSUPPORTED_QOS_POLICY`	18	Attempted to pass a QosPolicy setting that is not (yet) supported.
`ERRORCODE_CONTAINS_CONDITIONS`	19	Attempted to delete a WaitSet that still has Conditions attached to it.
`ERRORCODE_CONTAINS_LOANS`	20	Attempted to delete a DataReader/DataView that has unreturned loans.
`ERRORCODE_INCONSISTENT_TOPIC`	21	Attempted to create a topic that is inconsistent with existing topic definitions.

3.1.10.3 get_message

Scope

```
DDS.ErrorInfo
```

Synopsis

```
import DDS.*;

public int
  get_message
  (DDS.StringHolder message);
```
 or
```
public int
  get_message
  (org.omg.CORBA.StringHolder message);
```

Description

This operation retrieves the description of the latest error.

Two variants of the operation exist; the first can be used under all circumstances. The second is purely meant as a convenience when using the API in cohabitation with CORBA.

Parameters

in DDS.StringHolder message - The holder in which the latest error message will be stored.

or

in `org.omg.CORBA.StringHolder message` - The holder in which the latest error message will be stored.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_NO_DATA`.

Detailed Description

This operation stores the description of the latest error in a newly-allocated string. If the pointer supplied by the application through the message parameter already contains a string, it is freed. If no error has occurred, `RETCODE_NO_DATA` is returned and `null` is assigned to the value attribute in the message parameter.

3.1.10.4 **get_location**

Scope

DDS.ErrorInfo

Synopsis

```
public int
  get_location
  (DDS.StringHolder location);
```

or

```
public int
  get_location
  (org.omg.CORBA.StringHolder location);
```

Description

This operation retrieves the location or context of the latest error.

Two variants of the operation exist; the first can be used under all circumstances. The second is purely meant as a convenience when using the API in cohabitation with CORBA.

Parameters

in `DDS.StringHolder message` - The holder that has a value attribute where the string holding the location of the latest error will be stored.

or

in `org.omg.CORBA.StringHolder message` - The holder that has a value attribute where the string holding the location of the latest error will be stored.

Return Value

`int` - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_NO_DATA`.

Detailed Description

This operation stores the context or location of the latest error in a newly-allocated string. The string may contain the name of an operation or component of the data distribution service in which the error occurred, or other descriptive information on the location of the error. If the pointer supplied by the application through the location parameter already contains a string, it is freed. If no error has occurred, `RETCODE_NO_DATA` is returned andand `null` is assigned to the value in the location parameter.

3.1.10.5 **get_source_line**

Scope

   `DDS.ErrorInfo`

Synopsis

```
public int
  get_source_line
  (DDS.StringHolder sourceLine);
```

  or

```
public int
  get_source_line
  (org.omg.CORBA.StringHolder sourceLine);
```

Description

This operation retrieves the location within the sourcecode of the latest error.

Two variants of the operation exist; the first can be used under all circumstances. The second is purely meant as a convenience when using the API in cohabitation with CORBA.

Parameters

in DDS.StringHolder source_line - The holder in which the latest source line will be stored.

   or

in org.omg.CORBA.StringHolder source_line - The holder in which the latest source line will be stored.

Return Value

int - Possible return codes of the operation are:
    RETCODE_OK, RETCODE_NO_DATA.

Detailed Description

This operation stores the name and line number of the source file in which the latest error occurred, seperated by a colon, in a newly-allocated string in the source_line parameter. If the holder supplied by the application through the source_line parameter already contains a string, it is freed. If no error has occurred, RETCODE_NO_DATA is returned and null is assigned to the value within the source_line parameter.

3.1.10.6 **get_stack_trace**

Scope

    DDS.ErrorInfo

Synopsis

```
public int
  get_stack_trace
  (DDS.StringHolder stackTrace);
```
  or
```
public int
  get_stack_trace
  (org.omg.CORBA.StringHolder stackTrace);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.2 **Domain Module**
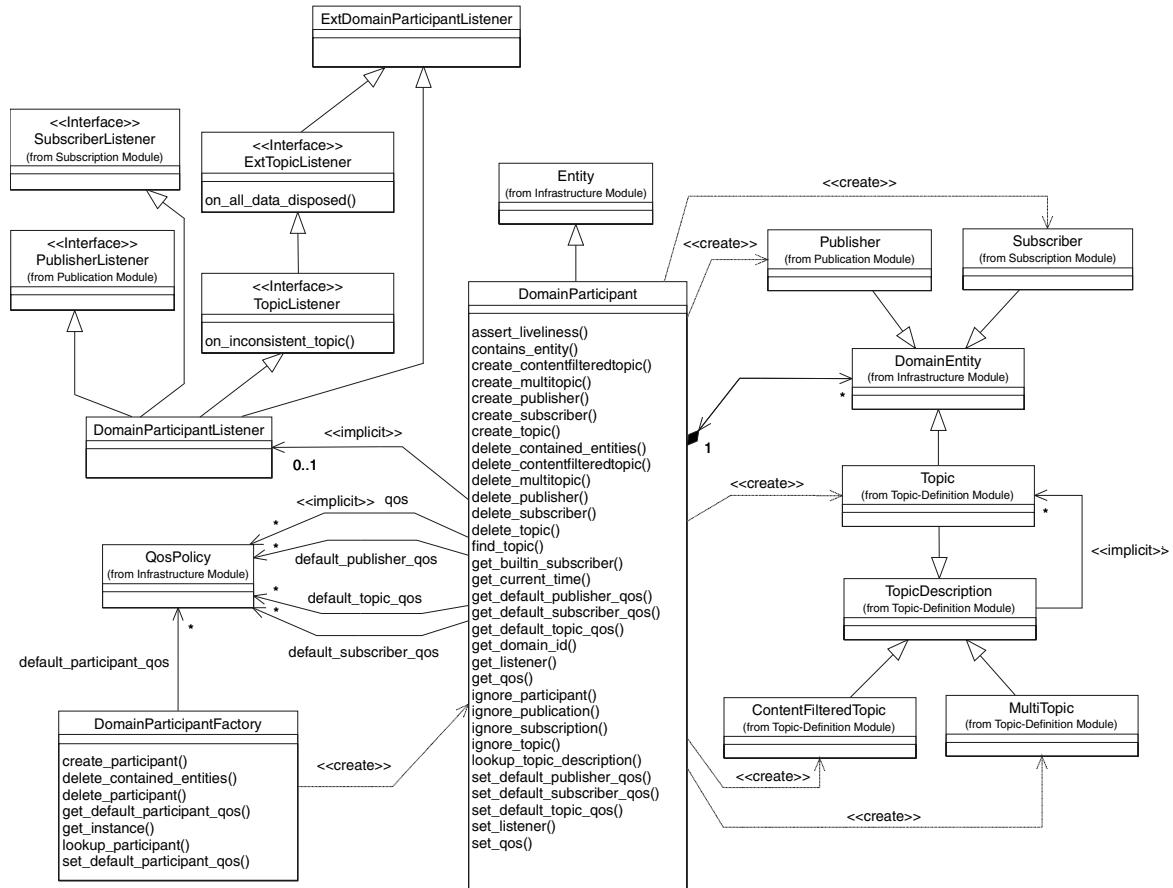
Figure 15 DCPS Domain Module's Class Model

This module contains the following classes:

- `DomainParticipant`
- `DomainParticipantFactory`
- `DomainParticipantListener` (interface)
- `Domain` (*not depicted*)

3.2.1 **Class DomainParticipant**

All the DCPS `Entity` objects are attached to a `DomainParticipant`.
A `DomainParticipant` represents the local membership of the application in a
`Domain`.

A `Domain` is a distributed concept that links all the applications that must be able to communicate with each other. It represents a communication plane: only the `Publishers` and the `Subscribers` attached to the same `Domain` can interact.

This class implements several functions:

- It acts as a container for all other `Entity` objects

- It acts as a factory for the `Publisher`, `Subscriber`, `Topic`, `ContentFilteredTopic` and `MultiTopic` objects

- It provides access to the built-in `Topic` objects

- It provides information about `Topic` objects

- It isolates applications within the same `Domain` (sharing the same `domainId`) from other applications in a different `Domain` on the same set of computers. 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, which allow the application to ignore locally any information about a given `Participant`, `Publication`, `Subscription` or `Topic`

The interface description of this class is as follows:

```
public interface DomainParticipant
{
//
// extends interface Entity
//
// public StatusCondition
//    get_statuscondition
//        (void);
// public int
//    get_status_changes
//        (void);
// public int
//    enable
//        (void);
//
// implemented API operations
//
   public Publisher
      create_publisher
         (PublisherQos qos,
          PublisherListener a_listener,
          int mask);
   public int
      delete_publisher
         (Publisher p);
   public Subscriber
```

```
                         create_subscriber
                            (SubscriberQos qos,
                            SubscriberListener a_listener,
                            int mask);
                      public int
                         delete_subscriber
                            (Subscriber s);
                      public Subscriber
                         get_builtin_subscriber
                            (void);
                      public Topic
                         create_topic
                            (String topic_name,
                            String type_name,
                            TopicQos qos,
                            TopicListener a_listener,
                            int mask);
                      public int
                         delete_topic
                            (Topic a_topic);
                      public Topic
                         find_topic
                            (String topic_name,
                            Duration_t timeout);
                      public TopicDescription
                         lookup_topicdescription
                            (String name);
                      public ContentFilteredTopic
                         create_contentfilteredtopic
                            (String name,
                            Topic related_topic,
                            String filter_expression,
                            String[] expression_parameters);
                      public int
                         delete_contentfilteredtopic
                            (ContentFilteredTopic a_contentfilteredtopic);
                      public MultiTopic
                         create_multitopic
                            (String name,
                            String type_name,
                            String subscription_expression,
                            String[] expression_parameters);
                      public int
                         delete_multitopic
                            (MultiTopic a_multitopic);
                      public int
                         delete_contained_entities
                            (void);
                      public int
                         set_qos
```

```
                            (DomainParticipantQos qos);
                public int
                    get_qos
                        (DomainParticipantQosHolder qos);
                public int
                    set_listener
                        (DomainParticipantListener a_listener,
                        int mask);
                public DomainParticipantListener
                    get_listener
                        (void);
                public int
                    ignore_participant
                        (long handle);
                public int
                    ignore_topic
                        (long handle);
                public int
                    ignore_publication
                        (long handle);
                public int
                    ignore_subscription
                        (long handle);
                public String
                    get_domain_id
                        (void);
                public int
                    get_discovered_participants
                        (InstanceHandleSeqHolder participant_handles);
                public int
                    get_discovered_participant_data
                        (long ParticipantBuiltinTopicDataHolder participant_data,
                            handle);
                public int
                    get_discovered_topics
                        (InstanceHandleSeqHolder topic_handles);
                public int
                    get_discovered_topic_data
                        (long TopicBuiltinTopicDataHolder topic_data,
                            handle);
                public int
                    assert_liveliness
                        (void);
                public int
                    set_default_publisher_qos
                        (PublisherQos qos);
                public int
                    get_default_publisher_qos
                        (PublisherQosHolder qos);
                public int
```

```
            set_default_subscriber_qos
                (SubscriberQos qos);
        public int
            get_default_subscriber_qos
                (SubscriberQosHolder qos);
        public int
            set_default_topic_qos
                (TopicQos qos);
        public int
            get_default_topic_qos
                (TopicQosHolder qos);
        public boolean
            contains_entity
                (long a_handle);
        public int
            get_current_time
                (Time_tHolder current_time);
    };
```

The following paragraphs describe the usage of all `DomainParticipant`
operations. The inherited operations are listed but not fully described because they
are not implemented in this class. The full description of these operations is given in
the classes from which they are inherited.

3.2.1.1 assert_liveliness

Scope

```
    DDS.DomainParticipant
```

Synopsis

```
    import DDS.*;
    public int
        assert_liveliness
            (void);
```

Description

This operation asserts the liveliness for the `DomainParticipant`.

Parameters

<none>

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,`
    `RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation will manually assert the liveliness for the `DomainParticipant`. This way, the Data Distribution Service is informed that the `DomainParticipant` is still alive. This operation only needs to be used when the `DomainParticipant` contains `DataWriters` with the `LivelinessQosPolicy` set to `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS`, and it will only affect the liveliness of those `DataWriters`.

Writing data via the `write` operation of a `DataWriter` will assert the liveliness on the `DataWriter` itself and its `DomainParticipant`. Therefore, `assert_liveliness` is only needed when *not* writing regularly.

The liveliness should be asserted by the application, depending on the `LivelinessQosPolicy`.

Return Code

When the operation returns:

- *RETCODE_OK* - the liveliness of this `DomainParticipant` has successfully been asserted.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `DomainParticipant` is not enabled.

3.2.1.2 contains_entity

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public boolean
   contains_entity
      (long a_handle);
```

Description

This operation checks whether or not the given `Entity` represented by `a_handle` is created by the `DomainParticipant` or any of its contained entities.

Parameters

in long a_handle - an `Entity` in the Data Distribution System.

Return Value

boolean - `true` if `a_handle` represents an `Entity` that is created by the `DomainParticipant` or any of its contained `Entities`. Otherwise the return value is `false`.

Detailed Description

This operation checks whether or not the given `Entity` represented by `a_handle` is created by the `DomainParticipant` itself (`TopicDescription`, `Publisher` or `Subscriber`) or created by any of its contained entities (`DataReader`, `ReadCondition`, `QueryCondition`, `DataWriter`, etc.).

Return value is `true` if `a_handle` represents an `Entity` that is created by the `DomainParticipant` or any of its contained `Entities`. Otherwise the return value is `false`.

3.2.1.3 create_contentfilteredtopic

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public ContentFilteredTopic
    create_contentfilteredtopic
        (String name,
          Topic related_topic,
          String filter_expression,
          String[] expression_parameters);
```

Description

This operation creates a `ContentFilteredTopic` for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a subset of the topic content.

Parameters

in String name - the name of the `ContentFilteredTopic`.

in Topic related_topic - the reference to the base topic on which the filtering will be applied. Therefore, a filtered topic is based on an existing Topic.

in String filter_expression - the SQL expression (subset of SQL), which defines the filtering.

PRISMTECH

>   *in String[] expression_parameters* - the handle to a sequence of strings
>   with the parameter value used in the SQL expression (i.e., the number of %n
>   tokens in the expression). The number of values in `expression_parameters`
>   must be equal or greater than the highest referenced %n token in the
>   `filter_expression` (e.g. if %1 and %8 are used as parameter in the
>   `filter_expression`, the `expression_parameters` should at least contain
>   n+1 = 9 values).

Return Value

ContentFilteredTopic - Return value is the reference to the newly created
ContentFilteredTopic. In case of an error, a `null` reference is returned.

Detailed Description

This operation creates a `ContentFilteredTopic` for a `DomainParticipant` in
order to allow `DataReaders` to subscribe to a subset of the topic content. The base
topic, which is being filtered is defined by the parameter `related_topic`. The
resulting `ContentFilteredTopic` only relates to the samples published under the
`related_topic`, which have been filtered according to their content. The resulting
`ContentFilteredTopic` only exists at the `DataReader` side and will never be
published. The samples of the `related_topic` are filtered according to the SQL
expression (which is a subset of SQL) as defined in the parameter
`filter_expression` (see Appendix H, *DCPS Queries and Filters*).

The `filter_expression` may also contain parameters, which appear as %n
tokens in the expression which must be set by the sequence of strings defined by the
parameter `expression_parameters`. The number of values in
`expression_parameters` must be equal or greater than the highest referenced
%n token in the `filter_expression` (e.g. if %1 and %8 are used as parameter in
the `filter_expression`, the `expression_parameters` should at least contain
n+1 = 9 values).

The `filter_expression` is a string that specifies the criteria to select the data
samples of interest. In other words, it identifies the selection of data from the
associated `Topics`. It is an SQL expression where the `WHERE` clause gives the
content filter.

3.2.1.4 create_multitopic

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public MultiTopic
```

```
create_multitopic
   (String name,
     String type_name,
     String subscription_expression,
     String[] expression_parameters);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation creates a `MultiTopic` for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a filtered/re-arranged combination and/or subset of the content of several `topics`.

Parameters

in String name - the name of the multi topic.

in String type_name - the name of the type of the `MultiTopic`. This `type_name` must have been registered using `register_type` prior to calling this operation.

in String subscription_expression - the SQL expression (subset of SQL), which defines the selection, filtering, combining and re-arranging of the sample data.

in String[] expression_parameters - the handle to a sequence of strings with the parameter value used in the SQL expression (i.e., the number of %n tokens in the expression). The number of values in `expression_parameters` must be equal or greater than the highest referenced %n token in the `subscription_expression` (e.g. if %1 and %8 are used as parameter in the `subscription_expression`, the `expression_parameters` should at least contain n+1 = 9 values).

Return Value

MultiTopic - Return value is the reference to the newly created `MultiTopic`. In case of an error, a `null` reference is returned.

Detailed Description

This operation creates a multiple topic for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a filtered/re-arranged combination and/or subset of the content of several `topics`.

Before the `MultiTopic` can be created, the `type_name` of the `MultiTopic` must have been registered prior to calling this operation. Registering is done, using the `register_type` operation from `TypeSupport`. The list of topics and the logic,

which defines the selection, filtering, combining and re-arranging of the sample data, is defined by the SQL expression (subset of SQL) defined in `subscription_expression`.

The `subscription_expression` may also contain parameters, which appear as `%n` tokens in the expression. These parameters are defined in `expression_parameters`. The number of values in expression_parameters must be equal or greater than the highest referenced `%n` token in the `subscription_expression` (e.g. if `%1` and `%8` are used as parameter in the `subscription_expression`, the `expression_parameters` should at least contain n+1 = 9 values).

The `subscription_expression` is a string that specifies the criteria to select the data samples of interest. In other words, it identifies the selection and rearrangement of data from the associated `Topics`. It is an SQL expression where the `SELECT` clause 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 `DataReader`, which is associated with a `MultiTopic` only accesses information which exist locally in the `DataReader`, based on the `Topics` used in the `subscription_expression`. The actual `MultiTopic` will never be produced, only the individual `Topics`.

3.2.1.5 create_publisher

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public Publisher
   create_publisher
      (PublisherQos qos,
        PublisherListener a_listener,
        int mask);
```

Description

This operation creates a `Publisher` with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `PublisherListener` to it.

Parameters

in PublisherQos qos - a collection of `QosPolicy` settings for the new `Publisher`. In case these settings are not self consistent, no `Publisher` is created.

in PublisherListener a_listener - a reference to the `PublisherListener` instance which will be attached to the new `Publisher`. It is permitted to use `null` as the value of the listener: this behaves as a `PublisherListener` whose operations perform no action.

in int mask - a bit-mask in which each bit enables the invocation of the `PublisherListener` for a certain status.

Return Value

Publisher - Return value is a reference to the newly created `Publisher`. In case of an error, the `null` reference is returned.

Detailed Description

This operation creates a `Publisher` with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `PublisherListener` to it. When the `PublisherListener` is not applicable, the `null` reference must be supplied instead. To delete the `Publisher` the operation `delete_publisher` or `delete_contained_entities` must be used.

In case the specified `QosPolicy` settings are not consistent, no `Publisher` is created and the `null` reference is returned. The `null` reference can also be returned when insufficient access rights exist for the partition(s) listed in the provided `QoS` structure.

Default QoS

The constant `PUBLISHER_QOS_DEFAULT` can be used as parameter `qos` to create a `Publisher` with the default `PublisherQos` as set in the `DomainParticipant`. The effect of using `PUBLISHER_QOS_DEFAULT` is the same as calling the operation `get_default_publisher_qos` and using the resulting `PublisherQos` to create the `Publisher`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `PublisherListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The

status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `PublisherListener`:

- `OFFERED_DEADLINE_MISSED_STATUS`   *(propagated)*
- `OFFERED_INCOMPATIBLE_QOS_STATUS`   *(propagated)*
- `LIVELINESS_LOST_STATUS`   *(propagated)*
- `PUBLICATION_MATCHED_STATUS`   *(propagated).*

⚠ Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataWriterListener` of a contained `DataWriter`, the `DataWriterListener` on that contained `DataWriter` is invoked instead of the `PublisherListener`. This means that a status change on a contained `DataWriter` only invokes the `PublisherListener` if the contained `DataWriter` itself does not handle the trigger event generated by the status change.

In case a communication status is not activated in the `mask` of the `PublisherListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Publisher` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

3.2.1.6 **create_subscriber**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public Subscriber
   create_subscriber
       (SubscriberQos qos,
         SubscriberListener a_listener,
         int mask);
```

Description

This operation creates a `Subscriber` with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `SubscriberListener` to it.

Parameters

in SubscriberQos qos - a collection of `QosPolicy` settings for the new `Subscriber`. In case these settings are not self consistent, no `Subscriber` is created.

in SubscriberListener a_listener - a reference to the `SubscriberListener` instance which will be attached to the new `Subscriber`. It is permitted to use `null` as the value of the listener: this behaves as a `SubscriberListener` whose operations perform no action.

in int mask - a bit-mask in which each bit enables the invocation of the `SubscriberListener` for a certain status.

Return Value

Subscriber - Return value is a reference to the newly created `Subscriber`. In case of an error, the `null` reference is returned.

Detailed Description

This operation creates a `Subscriber` with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `SubscriberListener` to it. When the `SubscriberListener` is not applicable, the `null` reference must be supplied instead. To delete the `Subscriber` the operation `delete_subscriber` or `delete_contained_entities` must be used.

In case the specified `QosPolicy` settings are not consistent, no `Subscriber` is created and the `null` reference is returned. The `null` reference can also be returned when insufficient access rights exist for the partition(s) listed in the provided `QoS` structure.

PRISMTECH

Default QoS

The constant `SUBSCRIBER_QOS_DEFAULT` can be used as parameter `qos` to create a `Subscriber` with the default `SubscriberQos` as set in the `Domainparticipant`. The effect of using `SUBSCRIBER_QOS_DEFAULT` is the same as calling the operation `get_default_subscriber_qos` and using the resulting `SubscriberQos` to create the `Subscriber`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `SubscriberListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `SubscriberListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS`   *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`   *(propagated)*
- `SAMPLE_LOST_STATUS`   *(propagated)*
- `SAMPLE_REJECTED_STATUS`   *(propagated)*
- `DATA_AVAILABLE_STATUS`   *(propagated)*
- `LIVELINESS_CHANGED_STATUS`   *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS`   *(propagated)*.
- `DATA_ON_READERS_STATUS`.

⚠ Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataReaderListener` of a contained `DataReader`, the `DataReaderListener` on that contained `DataReader` is invoked instead of the `SubscriberListener`. This means that a status change on a contained `DataReader` only invokes the `SubscriberListener` if the contained `DataReader` itself does not handle the trigger event generated by the status change.

In case a communication status is not activated in the `mask` of the `SubscriberListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Subscriber` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are "Read Communication Statuses" and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a DataReader, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

3.2.1.7 create_topic

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*
public Topic
   create_topic
      (String topic_name,
        String type_name,
        TopicQos qos,
        TopicListener a_listener,
        int mask);
```

PRISMTECH

Description

This operation creates a reference to a new or existing `Topic` under the given name, for a specific type, with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `TopicListener` to it.

Parameters

in `String topic_name` - the name of the `Topic` to be created. A new `Topic` will only be created, when no `Topic`, with the same name, is found within the `DomainParticipant`.

in `String type_name` - a local alias of the data type, which must have been registered before creating the `Topic`.

in `TopicQos qos` - a collection of `QosPolicy` settings for the new `Topic`. In case these settings are not self consistent, no `Topic` is created.

in `TopicListener a_listener` - a reference to the `TopicListener` instance which will be attached to the new `Topic`. It is permitted to use `null` as the value of the listener: this behaves as a `TopicListener` whose operations perform no action.

in `int mask` - a bit-mask in which each bit enables the invocation of the `TopicListener` for a certain status.

Return Value

`Topic` - Return value is a reference to the new or existing `Topic`. In case of an error, the `null` reference is returned.

Detailed Description

This operation creates a reference to a new or existing `Topic` under the given name, for a specific type, with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `TopicListener` to it. When the `TopicListener` is not applicable, the `null` reference must be supplied instead. In case the specified `QosPolicy` settings are not consistent, no `Topic` is created and the `null` reference is returned. To delete the `Topic` the operation `delete_topic` or `delete_contained_entities` must be used.

Default QoS

The constant `TOPIC_QOS_DEFAULT` can be used as parameter `qos` to create a `Topic` with the default `TopicQos` as set in the `DomainParticipant`. The effect of using `TOPIC_QOS_DEFAULT` is the same as calling the operation `get_default_topic_qos` and using the resulting `TopicQos` to create the `Topic`.

The `Topic` is bound to the type `type_name`. Prior to creating the `Topic`, the `type_name` must have been registered with the Data Distribution Service. Registering the `type_name` is done using the data type specific `register_type` operation.

Existing Topic Name

Before creating a new `Topic`, this operation performs a `lookup_topicdescription` for the specified `topic_name`. When a `Topic` is found with the same name in the current domain, the QoS and `type_name` of the found `Topic` are matched against the parameters `qos` and `type_name`. When they are the same, no `Topic` is created but a new proxy of the existing `Topic` is returned. When they are not exactly the same, no `Topic` is created and the `null` reference is returned.

When a `Topic` is obtained multiple times, it must also be deleted that same number of times using `delete_topic` or calling `delete_contained_entities` once to delete all the proxies.

Local Proxy

Since a `Topic` is a global concept in the system, access is provided through a local proxy. In other words, the reference returned is actually not a reference to a `Topic` but to a locally-created proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. For each create, a new proxy is created. Therefore the `Topic` must be deleted the same number of times, as the `Topic` was created with the same `topic_name` per Domain. In other words, each reference (local proxy) must be deleted separately.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `TopicListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `TopicListener`:

- `INCONSISTENT_TOPIC_STATUS`

The following statuses are applicable to the `ExtTopicListener`:

- `ALL_DATA_DISPOSED_TOPIC_STATUS`

NOTE: The `DDS.STATUS_MASK_ANY_V1_2` mask does not include the `ALL_DATA_DISPOSED_TOPIC_STATUS` bit, because this is an OpenSplice extension.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

In case a communication status is not activated in the `mask` of the `TopicListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Topic` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

3.2.1.8 **delete_contained_entities**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   delete_contained_entities
      (void);
```

Description

This operation deletes all the `Entity` objects that were created on the `DomainParticipant`.

Parameters

\<none>

Return Value

int - Possible return codes of the operation are:
    RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
    RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation deletes all the `Entity` objects that were created on the `DomainParticipant`. In other words, it deletes all `Publisher`, `Subscriber`, `Topic`, `ContentFilteredTopic` and `MultiTopic` objects. Prior to deleting each contained `Entity`, this operation regressively calls the corresponding `delete_contained_entities` operation on each `Entity` (if applicable). In other words, all `Entity` objects in the `Publisher` and `Subscriber` are deleted, including the `DataWriter` and `DataReader`. Also the `QueryCondition` and `ReadCondition` objects contained by the `DataReader` are deleted.

Topic

Since a `Topic` is a global concept in the system, access is provided through a local proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. Such a proxy is created by the `create_topic` or `find_topic` operation. When a reference to the same `Topic` was created multiple times (either by `create_topic` or `find_topic`), all references (local proxies) are deleted. With the last proxy, the `Topic` itself is also removed from the system.

⚠️ **NOTE***:* The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained `DataReader` cannot be deleted because the application has called a `read` or `take` operation and has not called the corresponding `return_loan` operation to return the loaned samples. In such cases, the operation does not roll back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

- *RETCODE_OK* - the contained `Entity` objects are deleted and the application may delete the `DomainParticipant`.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.

PRISMTECH

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - one or more of the contained entities are in a state where they cannot be deleted.

3.2.1.9 delete_contentfilteredtopic

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   delete_contentfilteredtopic
      (ContentFilteredTopic a_contentfilteredtopic);
```

Description

This operation deletes a ContentFilteredTopic.

Parameters

in ContentFilteredTopic a_contentfilteredtopic - a reference to the ContentFilteredTopic, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation deletes a ContentFilteredTopic.

The deletion of a ContentFilteredTopic is not allowed if there are any existing DataReader objects that are using the ContentFilteredTopic. If the delete_contentfilteredtopic operation is called on a ContentFilteredTopic with existing DataReader objects attached to it will return PRECONDITION_NOT_MET.

The delete_contentfilteredtopic operation must be called on the same DomainParticipant object used to create the ContentFilteredTopic. If delete_contentfilteredtopic is called on a different DomainParticipant the operation will have no effect and it will return PRECONDITION_NOT_MET.

<u>*Return Code*</u>

When the operation returns:

- *RETCODE_OK* - the `ContentFilteredTopic` is deleted

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter `a_contentfilteredtopic` is not a valid `ContentFilteredTopic` reference.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DomainParticipant`, as used when the `ContentFilteredTopic` was created, or the `ContentFilteredTopic` is being used by one or more `DataReader` objects.

3.2.1.10 delete_multitopic

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   delete_multitopic
       (MultiTopic a_multitopic);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation deletes a `MultiTopic`.

Parameters

in MultiTopic a_multitopic - a reference to the `MultiTopic`, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or
RETCODE_PRECONDITION_ NOT_MET.
```

PRISMTECH

Detailed Description

This operation deletes a `MultiTopic`.

The deletion of a `MultiTopic` is not allowed if there are any existing `DataReader` objects that are using the `MultiTopic`. If the `delete_multitopic` operation is called on a `MultiTopic` with existing `DataReader` objects attached to it will return `RETCODE_PRECONDITION_NOT_MET`.

The `delete_multitopic` operation must be called on the same `DomainParticipant` object used to create the `MultiTopic`. If `delete_multitopic` is called on a different `DomainParticipant` the operation will have no effect and it will return `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `MultiTopic` is deleted
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `a_multitopic` is not a valid `MultiTopic` reference.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DomainParticipant`, as used when the `MultiTopic` was created, or the `MultiTopic` is being used by one or more `DataReader` objects.

3.2.1.11 delete_publisher

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   delete_publisher
      (Publisher p);
```

Description

This operation deletes a `Publisher`.

Parameters

in Publisher p - a reference to the `Publisher`, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
   RETCODE_ALREADY_ DELETED, RETCODE_OUT_OF_RESOURCES or
   RETCODE_PRECONDITION_ NOT_MET.

Detailed Description

This operation deletes a `Publisher`. A `Publisher` cannot be deleted when it has any attached `DataWriter` objects. When the operation is called on a `Publisher` with DataWriter objects, the operation returns RETCODE_PRECONDITION_NOT_MET. When the operation is called on a different `DomainParticipant`, as used when the `Publisher` was created, the operation has no effect and returns RETCODE_PRECONDITION_NOT_MET.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Publisher` is deleted

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter p is not a valid `Publisher` reference.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DomainParticipant`, as used when the `Publisher` was created, or the `Publisher` contains one or more `DataWriter` objects.

3.2.1.12 delete_subscriber

Scope

   DDS.DomainParticipant

Synopsis

```
import DDS.*
public int
   delete_subscriber
```

```
(Subscriber s);
```

Description

This operation deletes a `Subscriber`.

Parameters

in Subscriber s - a reference to the `Subscriber`, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
  `RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,`
  `RETCODE_ALREADY_ DELETED, RETCODE_OUT_OF_RESOURCES` or
  `RETCODE_PRECONDITION_ NOT_MET.`

Detailed Description

This operation deletes a `Subscriber`. A `Subscriber` cannot be deleted when it has any attached `DataReader` objects. When the operation is called on a `Subscriber` with `DataReader` objects, the operation returns `RETCODE_PRECONDITION_NOT_MET`. When the operation is called on a different `DomainParticipant`, as used when the `Subscriber` was created, the operation has no effect and returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Subscriber` is deleted

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter `s` is not a valid `Subscriber` reference.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DomainParticipant`, as used when the `Subscriber` was created, or the `Subscriber` contains one or more `DataReader` objects.

3.2.1.13 delete_topic

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
    delete_topic
        (Topic a_topic);
```

Description

This operation deletes a `Topic`.

Parameters

in Topic a_topic - a reference to the `Topic`, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_ DELETED, RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_ NOT_MET.

Detailed Description

This operation deletes a `Topic`. A `Topic` cannot be deleted when there are any `DataReader`, `DataWriter`, `ContentFilteredTopic` or `MultiTopic` objects, which are using the `Topic`. When the operation is called on a `Topic` referenced by any of these objects, the operation returns RETCODE_PRECONDITION_NOT_MET. When the operation is called on a different `DomainParticipant`, as used when the `Topic` was created, the operation has no effect and returns RETCODE_PRECONDITION_NOT_MET.

Local Proxy

Since a `Topic` is a global concept in the system, access is provided through a local proxy. In other words, the reference is actually not a reference to a `Topic` but to the local proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. Such a proxy is created by the `create_topic` or `find_topic` operation. This operation will delete the local proxy. When a reference to the same `Topic` was created multiple times (either by `create_topic` or `find_topic`), each reference (local proxy) must be deleted separately. When this proxy is the last proxy for this `Topic`, the `Topic` itself is also removed from the system. As mentioned, a proxy may only be deleted when there are no other entities attached to it. However, it is possible to delete a proxy while there are entities attached to a different proxy.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Topic` is deleted
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `a_topic` is not a valid `Topic` reference.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DomainParticipant`, as used when the `Topic` was created, or the `Topic` is still referenced by other objects.

3.2.1.14 **enable (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   enable
      (void);
```

3.2.1.15 **find_topic**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public Topic
   find_topic
      (String topic_name,
        Duration_t timeout);
```

Description

This operation gives access to an existing (or ready to exist) enabled `Topic`, based on its `topic_name`.

Parameters

in String topic_name - the name of the `Topic` that the application wants access to.

in Duration_t timeout - the maximum duration to block for the `find_topic`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

Topic - a reference to the `Topic` found.

Detailed Description

This operation gives access to an existing `Topic`, based on its `topic_name`. The operation takes as arguments the `topic_name` of the `Topic` and a `timeout`.

If a `Topic` of the same `topic_name` already exists, it gives access to this `Topic`. Otherwise it waits (blocks the caller) until another mechanism creates it. This other mechanism can be another thread, a configuration tool, or some other Data Distribution Service utility. If after the specified `timeout` the `Topic` can still not be found, the caller gets unblocked and the `null` reference is returned.

A `Topic` obtained by means of `find_topic`, must also be deleted by means of `delete_topic` so that the local resources can be released. If a `Topic` is obtained multiple times it must also be deleted that same number of times using `delete_topic` or calling `delete_contained_entities` once to delete all the proxies.

A `Topic` that is obtained by means of `find_topic` in a specific `DomainParticipant` can only be used to create `DataReaders` and `DataWriters` in that `DomainParticipant` if its corresponding `TypeSupport` has been registered to that same `DomainParticipant`.

Local Proxy

Since a `Topic` is a global concept in the system, access is provided through a local proxy. In other words, the reference returned is actually not a reference to a `Topic` but to a locally-created proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. For each time this operation is called, a new proxy is created. Therefore the `Topic` must be deleted the same number of times, as the `Topic` was created with the same `topic_name` per Domain. In other words, each reference (local proxy) must be deleted separately.

3.2.1.16 get_builtin_subscriber

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public Subscriber
   get_builtin_subscriber
       (void);
```

Description

This operation returns the built-in `Subscriber` associated with the `DomainParticipant`.

Parameters

<none>

Return Value

Subscriber - Result value is a reference to the built-in `Subscriber` associated with the `DomainParticipant`.

Detailed Description

This operation returns the built-in `Subscriber` associated with the `DomainParticipant`. Each `DomainParticipant` contains several built-in `Topic` objects. The built-in `Subscriber` contains the corresponding `DataReader` objects to access them. All these `DataReader` objects belong to a single built-in `Subscriber`. Note that there is exactly one built-in `Subscriber` associated with each `DomainParticipant`.

3.2.1.17 get_current_time

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   get_current_time
       (Time_tHolder current_time);
```

Description

This operation returns the value of the current time that the Data Distribution Service uses to time-stamp written data as well as received data in `current_time`.

Parameters

inout Time_tHolder current_time - the value of the current time as used by the Data Distribution System. The input value of `current_time` is ignored.

Return Value

int - Possible return codes of the operation are:
    RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
    RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or
    RETCODE_NOT_ENABLED.

Detailed Description

This operation returns the value of the current time that the Data Distribution
Service uses to time-stamp written data as well as received data in current_time.
The input value of current_time is ignored by the operation.

Return Code

When the operation returns:

- *RETCODE_OK* - the value of the current time is returned in current_time.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter current_time is not a valid
  reference.

- *RETCODE_ALREADY_DELETED* - the DomainParticipant has already been
  deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the DomainParticipant is not enabled.

3.2.1.18 **get_default_publisher_qos**

Scope

    DDS.DomainParticipant

Synopsis

```
import DDS.*;
public int
    get_default_publisher_qos
        (PublisherQosHolder qos);
```

Description

This operation gets an object with the default Publisher QosPolicy settings of
the DomainParticipant.

Parameters

inout PublisherQosHolder qos - a reference to the destination
    PublisherQosHolder object in which the default QosPolicy settings for the
    Publisher are written.

Return Value

int - Possible return codes of the operation are:
    RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or
    RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation gets an object of the class PublisherQos with the default
Publisher QosPolicy settings of the DomainParticipant (that is the
PublisherQos) which is used for newly created Publisher objects, in case the
constant PUBLISHER_QOS_DEFAULT is used. The default PublisherQos is only
used when the constant is supplied as parameter qos to specify the PublisherQos
in the create_publisher operation. The application must provide the
PublisherQos object in which the QosPolicy settings can be stored and pass the
qos reference to the operation. The operation writes the default QosPolicy
settings to the object referenced to by qos. Any settings in the object are
overwritten.

The values retrieved by this operation match the set of values specified on the last
successful call to set_default_publisher_qos, or, if the call was never made,
the default values as specified for each QosPolicy setting as defined in Table 2,
DCPS Holder Classes, on page 12.

Return Code

When the operation returns:

• *RETCODE_OK* - the default Publisher QosPolicy settings of this
  DomainParticipant have successfully been copied into the specified
  PublisherQosHolder parameter.

• *RETCODE_ERROR* - an internal error has occurred.

• *RETCODE_ALREADY_DELETED* - the DomainParticipant has already been
  deleted.

• *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.2.1.19 **get_default_subscriber_qos**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   get_default_subscriber_qos
      (SubscriberQosHolder qos);
```

Description

This operation gets an object with the default `Subscriber QosPolicy` settings of the `DomainParticipant`.

Parameters

inout SubscriberQosHolder qos - a reference to the destination `SubscriberQosHolder` object in which the default `QosPolicy` settings for the `Subscriber` are written.

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets an object of the class `SubscriberQos` with the default `Subscriber QosPolicy` settings of the `DomainParticipant` (that is the `SubscriberQos`) which is used for newly created `Subscriber` objects, in case the constant `SUBSCRIBER_QOS_DEFAULT` is used. The default `SubscriberQos` is only used when the constant is supplied as parameter `qos` to specify the `SubscriberQos` in the `create_subscriber` operation. The application must provide the QoS object in which the policy can be stored and pass the `qos` reference to the operation. The operation writes the default `QosPolicy` to the object referenced to by `qos`. Any settings in the object are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_subscriber_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` as defined in Table 2, *DCPS Holder Classes*, on page 12.

Return Code

When the operation returns:

- *RETCODE_OK* - the default `Subscriber QosPolicy` settings of this `DomainParticipant` have successfully been copied into the specified `SubscriberQosHolder` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.20 get_default_topic_qos

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
  get_default_topic_qos
      (TopicQosHolder qos);
```

Description

This operation gets an object with the default `Topic QosPolicy` settings of the `DomainParticipant`.

Parameters

inout TopicQosHolder qos - a reference to the destination `TopicQosHolder` object in which the default `QosPolicy` settings for the `Topic` are written.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets an object of the class `TopicQos` with the default `Topic QosPolicy` settings of the `DomainParticipant` (that is the `TopicQos`) which is used for newly created `Topic` objects, in case the constant `TOPIC_QOS_DEFAULT` is used. The default `TopicQos` is only used when the constant is supplied as parameter qos to specify the `TopicQos` in the `create_topic` operation. The application must provide an object of the `TopicQos` class in which the policy can

be stored and pass the `qos` reference to the operation. The operation writes the default `QosPolicy` settings to the object referenced to by `qos`. Any settings in the object are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_topic_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` as defined in Table 2, *DCPS Holder Classes*, on page 12.

Return Code

When the operation returns:

- *RETCODE_OK* - the default `Topic` `QosPolicy` settings of this `DomainParticipant` have successfully been copied into the specified `TopicQosHolder` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.21 **get_discovered_participants**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   get_discovered_participants
       (InstanceHandleSeqHolder participant_handles);
```

Description

This operation retrieves the list of `DomainParticipants` that have been discovered in the domain.

Parameters

inout InstanceHandleSeqHolder participant_handles - a sequence which is used to pass the list of all associated participants.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ILLEGAL_OPERATION`,
`RETCODE_UNSUPPORTED`,   `RETCODE_ALREADY_DELETED`,
`RETCODE_OUT_OF_RESOURCES`, or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation retrieves the list of `DomainParticipants` that have been
discovered in the domain and that the application has not indicated should be
"ignored" by means of the `DomainParticipant ignore_participant`
operation. The `participant_handles` sequence and its buffer may be
pre-allocated by the application and therefore must either be re-used in a subsequent
invocation of the `get_discovered_participants` operation or be released by
calling `free` on the returned `participant_handles`. If the pre-allocated
sequence is not big enough to hold the number of associated participants, the
sequence will automatically be (re-)allocated to fit the required size. The handles
returned in the `participant_handles` sequence are the ones that are used by the
DDS implementation to locally identify the corresponding matched `Participant`
entities. You can access more detailed information about a particular participant by
passing   its   `participant_handle`   to   the
`get_discovered_participant_data` operation.

Return Code

When the operation returns:

- *RETCODE_OK* - the list of associated participants has been successfully obtained.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ILLEGAL_OPERATION* - the operation is invoked on an inappropriate object.
- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" participants.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `DomainParticipant` is not enabled.

3.2.1.22 **get_discovered_participant_data**

Scope

`DDS.DomainParticipant`

Synopsis

```
import DDS.*;
public int
   get_discovered_participant_data
       (long ParticipantBuiltinTopicDataHolder participant_data,
         long participant_handle);
```

Description

This operation retrieves information on a `DomainParticipant` 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 `DomainParticipant ignore_participant` operation.

Parameters

inout ParticipantBuiltinTopicDataHolder participant_data - a pointer to the sample in which the information about the specified partition is to be stored.

in long participant_handle - a handle to the participant whose information needs to be retrieved.

Return Value

ReturnCode_t - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ILLEGAL_OPERATION, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES`, or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation retrieves information on a `DomainParticipant` 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 `DomainParticipant ignore_participant` operation.

The `partition_handle` must correspond to a partition currently associated with the `DomainParticipant`, otherwise the operation will fail and return `RETCODE_ERROR`. The operation `get_discovered_participant_data` can be used to find more detailed information about a particular participant that is found with the `get_discovered_participants` operation.

Return Code

When the operation returns:

- *RETCODE_OK* - the information on the specified partition has been successfully retrieved.

PRISMTECH

- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ILLEGAL_OPERATION* - the operation is invoked on an inappropriate object.
- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" partition.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `DomainParticipant` is not enabled.

3.2.1.23 get_discovered_topics

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
  get_discovered_topics
      (InstanceHandleSeqHolder topic_handles);
```

Description

This operation retrieves the list of `Topics` that have been discovered in the domain.

Parameters

inout InstanceHandleSeqHolder topic_handles - a sequence which is used to pass the list of all associated topics.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ILLEGAL_OPERATION, RETCODE_UNSUPPORTED,   RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, or RETCODE_NOT_ENABLED.

Detailed Description

This operation retrieves the list of `Topics` that have been discovered in the domain and that the application has not indicated should be "ignored" by means of the `DomainParticipant ignore_topic` operation. The `topic_handles` sequence and its buffer may be pre-allocated by the application and therefore must either be re-used in a subsequent invocation of the `get_discovered_topics` operation or

be released by calling free on the returned `topic_handles`. If the pre-allocated sequence is not big enough to hold the number of associated participants, the sequence will automatically be (re-)allocated to fit the required size. The handles returned in the `topic_handles` sequence are the ones that are used by the DDS implementation to locally identify the corresponding matched `Topic` entities. You can access more detailed information about a particular topic by passing its `topic_handle` to the `get_discovered_topic_data` operation.

Return Code

When the operation returns:

- *RETCODE_OK* - the list of associated topics has been successfully obtained.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ILLEGAL_OPERATION* - the operation is invoked on an inappropriate object.
- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" topics.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `DomainParticipant` is not enabled.

3.2.1.24 get_discovered_topic_data

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   get_discovered_topic_data
      (long TopicBuiltinTopicDataHolder topic_data,
       long topic_handle);
```

Description

This operation retrieves information on a `Topic` that has been discovered on the network. The topic must have been created by a participant in the same domain as the participant on which this operation is invoked and must not have been "ignored" by means of the `DomainParticipant ignore_topic` operation.

Parameters

inout TopicBuiltinTopicDataHolder topic_data - a pointer to the
sample in which the information about the specified topic is to be stored.

in long topic_handle - a handle to the topic whose information needs to be
retrieved.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ILLEGAL_OPERATION,
RETCODE_UNSUPPORTED,   RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES, or RETCODE_NOT_ENABLED.

Detailed Description

This operation retrieves information on a Topic that has been discovered on the
network. The topic must have been created by a participant in the same domain as
the participant on which this operation is invoked and must not have been "ignored"
by means of the DomainParticipant ignore_topic operation. The
topic_handle must correspond to a topic currently associated with the
DomainParticipant, otherwise the operation will fail and return
RETCODE_ERROR. The operation get_discovered_topic_data can be used to
find more detailed information about a particular topic that is found with the
get_discovered_topics operation.

Return Code

When the operation returns:

- *RETCODE_OK* - the information on the specified topic has successfully been
retrieved.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ILLEGAL_OPERATION* - the operation is invoked on an inappropriate
object.

- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the
information about "associated" topics.

- *RETCODE_ALREADY_DELETED* - the DomainParticipant has already been
deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the DomainParticipant is not enabled.

3.2.1.25 get_domain_id

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   get_domain_id
       (void);
```

Description

This operation returns the `DomainId` of the `Domain` to which this `DomainParticipant` is attached.

Parameters

<none>

Return Value

int - result is the `DomainId`.

Detailed Description

This operation returns the DomainId of the Domain to which this DomainParticipant is attached. Also see the create_participant operation.

3.2.1.26 get_listener

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public DomainParticipantListener
   get_listener
       (void);
```

Description

This operation allows access to a `DomainParticipantListener`.

Parameters

Return Value

DomainParticipantListener - result is a reference to the
   DomainParticipantListener attached to the DomainParticipant.

Detailed Description

This operation allows access to a DomainParticipantListener attached to the
DomainParticipant. When no DomainParticipantListener was attached to
the DomainParticipant, the null reference is returned.

3.2.1.27 **get_qos**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   get_qos
      (DomainParticipantQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a
DomainParticipant.

Parameters

inout DomainParticipantQosHolder qos - a reference to the destination
   DomainParticipantQosHolder object in which the QosPolicy settings
   will be copied.

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or
   RETCODE_OUT_OF_RESOURCES.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this
  DomainParticipant has successfully been copied into the specified
  DomainParticipantQosHolder parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

Detailed Description

This operation allows access to the existing set of QoS policies of a `DomainParticipant` on which this operation is used. This `DomainparticipantQos` is stored at the location referenced by the `qos` parameter.

3.2.1.28 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
       (void);
```

3.2.1.29 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
       (void);
```

3.2.1.30 ignore_participant

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*
public int
   ignore_participant
       (long handle);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.31 **ignore_publication**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
    ignore_publication
        (long handle);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.32 **ignore_subscription**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
    ignore_subscription
        (long handle);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.33 **ignore_topic**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
    ignore_topic
        (long handle);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.34 **lookup_topicdescription**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public TopicDescription
```

```
lookup_topicdescription
    (String name);
```

Description

This operation gives access to a locally-created `TopicDescription`, with a matching `name`.

Parameters

in String name - the name of the `TopicDescription` to look for.

Return Value

TopicDescription - Return value is a reference to the `TopicDescription` found. When no such `TopicDescription` is found, the `null` reference is returned.

Detailed Description

The operation `lookup_topicdescription` gives access to a locally-created `TopicDescription`, based on its name. The operation takes as argument the name of the `TopicDescription`.

If one or more local `TopicDescription` proxies (see also Section 3.2.1.15, *find_topic*, on page 163) of the same name already exist, a reference to one of the already existing local proxies is returned: `lookup_topicdescription` will never create a new local proxy. That means that the proxy that is returned does not need to be deleted separately from its original. When no local proxy exists, it returns the `null` reference. The operation never blocks.

The operation `lookup_topicdescription` may be used to locate any locally-created `Topic`, `ContentFilteredTopic`, and `MultiTopic` object.

3.2.1.35 set_default_publisher_qos

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
    set_default_publisher_qos
        (PublisherQos qos);
```

Description

This operation sets the default `PublisherQos` of the `DomainParticipant`.

PRISMTECH

Parameters

in `PublisherQos qos` - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created `Publishers`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES.`

Detailed Description

This operation sets the default `PublisherQos` of the `DomainParticipant` (that is the compound class with the `QosPolicy` settings) which is used for newly created `Publisher` objects, in case the constant `PUBLISHER_QOS_DEFAULT` is used. The default `PublisherQos` is only used when the constant is supplied as parameter `qos` to specify the `PublisherQos` in the `create_publisher` operation. The `PublisherQos` is always self consistent, because its policies do not depend on each other. This means that this operation never returns the `RETCODE_INCONSISTENT_POLICY`. The values set by this operation are returned by `get_default_publisher_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `PublisherQos` is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid `PublisherQos`.
- *RETCODE_UNSUPPORTED* - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.36 **set_default_subscriber_qos**

Scope

`DDS.DomainParticipant`

Synopsis

`import DDS.*;`

```
public int
  set_default_subscriber_qos
     (SubscriberQos qos);
```

Description

This operation sets the default `SubscriberQos` of the `DomainParticipant`.

Parameters

in SubscriberQos qos - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created `Subscribers`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation sets the default `SubscriberQos` of the `DomainParticipant` (that is the compound class with the `QosPolicy` settings) which is used for newly created `Subscriber` objects, in case the constant `SUBSCRIBER_QOS_DEFAULT` is used. The default `SubscriberQos` is only used when the constant is supplied as parameter qos to specify the `SubscriberQos` in the `create_subscriber` operation. The `SubscriberQos` is always self consistent, because its policies do not depend on each other. This means that this operation never returns the `RETCODE_INCONSISTENT_POLICY`. The values set by this operation are returned by `get_default_subscriber_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `SubscriberQos` is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid `PublisherQos`.
- *RETCODE_UNSUPPORTED* - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

PRISMTECH

3.2.1.37 **set_default_topic_qos**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
    set_default_topic_qos
        (TopicQos qos);
```

Description

This operation sets the default `TopicQos` of the `DomainParticipant`.

Parameters

in `TopicQos qos` - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created `Topics`.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_INCONSISTENT_POLICY`.

Detailed Description

This operation sets the default `TopicQos` of the `DomainParticipant` (that is the compound class with the `QosPolicy` settings) which is used for newly created `Topic` objects, in case the constant `TOPIC_QOS_DEFAULT` is used. The default `TopicQos` is only used when the constant is supplied as parameter `qos` to specify the `TopicQos` in the `create_topic` operation. This operation checks if the `TopicQos` is self consistent. If it is not, the operation has no effect and returns `RETCODE_INCONSISTENT_POLICY`. The values set by this operation are returned by `get_default_topic_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `TopicQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter `qos` is not a valid `TopicQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value.

- *RETCODE_UNSUPPORTED* - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_INCONSISTENT_POLICY* - the parameter `qos` contains conflicting `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.2.1.38 **set_listener**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   set_listener
      (DomainParticipantListener a_listener,
         int mask);
```

Description

This operation attaches a `DomainParticipantListener` to the `DomainParticipant`.

Parameters

in DomainParticipantListener a_listener - a reference to the `DomainParticipantListener` instance, which will be attached to the `DomainParticipant`.

in int mask - a bit mask in which each bit enables the invocation of the `DomainParticipantListener` for a certain status.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK,   RETCODE_ERROR,   RETCODE_UNSUPPORTED,` `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_ OF_RESOURCES`.

Detailed Description

This operation attaches a `DomainParticipantListener` to the `DomainParticipant`. Only one `DomainParticipantListener` can be attached to each `DomainParticipant`. If a `DomainParticipantListener` was already attached, the operation will replace it with the new one. When `a_listener` is the `null` reference, it represents a listener that is treated as a NOOP[1] for all statuses activated in the bitmask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `DomainParticipantListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset. An exception to this rule is the `null` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `DomainParticipantListener`:

- `INCONSISTENT_TOPIC_STATUS` *(propagated)*
- `OFFERED_DEADLINE_MISSED_STATUS` *(propagated)*
- `REQUESTED_DEADLINE_MISSED_STATUS` *(propagated)*
- `OFFERED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `SAMPLE_LOST_STATUS` *(propagated)*
- `SAMPLE_REJECTED_STATUS` *(propagated)*
- `DATA_ON_READERS_STATUS` *(propagated)*
- `DATA_AVAILABLE_STATUS` *(propagated)*
- `LIVELINESS_LOST_STATUS` *(propagated)*
- `LIVELINESS_CHANGED_STATUS` *(propagated)*
- `PUBLICATION_MATCHED_STATUS` *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS` *(propagated).*

Be aware that the `PUBLICATION_MATCHED_STATUS` and `SUBSCRIPTION_MATCHED_STATUS` are not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See

---

1. Short for **No-Op**eration, an instruction that peforms nothing at all.

the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `Listener` of a contained entity, the `Listener` on that contained entity is invoked instead of the `DomainParticipantListener`. This means that a status change on a contained entity only invokes the `DomainParticipantListener` if the contained entity itself does not handle the trigger event generated by the status change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are "Read Communication Statuses" and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a DataReader, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipant Listener` for the `DATA_AVAILABLE_STATUS` (in that order).

Return Code

When the operation returns:

- *RETCODE_OK* - the `DomainParticipantListener` is attached

- *RETCODE_ERROR* - an internal error has occurred

- *RETCODE_UNSUPPORTED* - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

PRISMTECH

3.2.1.39 **set_qos**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
public int
   set_qos
      (DomainParticipantQos qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipant`.

Parameters

in DomainParticipantQos qos - the new set of `QosPolicy` settings for the `DomainParticipant`.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipant`. The parameter `qos` contains the object with the `QosPolicy` settings which is checked for self-consistency.

The set of `QosPolicy` settings specified by the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned `RETCODE_OK`).

Return Code

When the operation returns:

- *RETCODE_OK* - the new `DomainParticipantQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.40 **delete_historical_data**

Scope

```
DDS.DomainParticipant
```

Synopsis

```
import DDS.*;
int
   delete_historical_data
      (String partition_expression,
       String topic_expression);
```

Description

This operation deletes all historical TRANSIENT and PERSISTENT data that is storedby the durability service that is configured to support this DomainParticipant.

Parameters

in String partition_expression - An expression to define a filter on partitions.

in String topic_expression - An expression to define a filter on topic names.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR.

Detailed Description

This operation deletes all historical TRANSIENT and PERSISTENT data that is stored by the durability service that is configured to support this DomainParticipant. It only deletes the samples stored in the transient and persistent store, samples stored in individual application DataReaders is spared and remains available to these readers. However, late-joiners will no longer be able to obtain the deleted samples.

The partition_expression and topic_expression strings can be used to specify selection criteria for the topic and/or partition in which the data will be deleted. Wildcards are supported. Note that these parameters are mandatory and cannot be empty. The "*" expression can be used to match all partitions and/or topics.

Only data that exists prior to this method invocationis deleted. Data that is still being inserted during this method invocationwill not be removed.

Return Code

When the operation returns:

- *RETCODE_OK* - all data matching the topic and partition expressions has been deleted.

- *RETCODE_ERROR* - an internal error has occurred.

3.2.2 Class DomainParticipantFactory

The purpose of this class is to allow the creation and destruction of DomainParticipant objects. DomainParticipantFactory itself has no factory. It is a pre-existing singleton object that can be accessed by means of the get_instance operation on the DomainParticipantFactory object.

The pre-defined value TheParticipantFactory can also be used as an alias for the singleton factory returned by the operation get_instance.

The interface description of this class is as follows:

```
 public class DomainParticipantFactory
{
//
// implemented API operations
//
    public static DomainParticipantFactory
       get_instance
           (void);
    public DomainParticipant
       create_participant
           (int domainId,
           DomainParticipantQos qos,
           DomainParticipantListener a_listener,
           int mask);
    public int
       delete_participant
           (DomainParticipant a_participant);
    public DomainParticipant
       lookup_participant
           (int domainId);
    public int
       set_default_participant_qos
           (DomainParticipantQos qos);
    public int
       get_default_participant_qos
           (DomainParticipantQosHolder qos);
    public int
       set_qos
           (DomainParticipantFactoryQos qos);
    public int
       get_qos
```

```
            (DomainParticipantFactoryQosHolder qos);
        public int
           delete_domain
              (Domain a_domain);
        public Domain
           lookup_domain
              (int domainId);
        public int
           delete_contained_entities
               (void);
        public int
           detach_all_domains
               (boolean block_operations,
               boolean delete_entities);
    };
```

The following paragraphs describe the usage of all `DomainParticipantFactory` operations.

3.2.2.1 create_participant

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
public DomainParticipant
   create_participant
       (int domainId,
         DomainParticipantQos qos,
         DomainParticipantListener a_listener,
         int mask);
```

Description

This operation creates a new `DomainParticipant` which will join the `domain` identified by `domainId`, with the desired `DomainParticipantQos` and attaches the optionally specified `DomainParticipantListener` to it.

Parameters

in int domainId - the ID of the Domain to which the `DomainParticipant` is joined. This should be the ID as specified in the configuration file. This will also be applicable for the `lookup_participant`, `lookup_domain` and `get_domain_id` operations.

in DomainParticipantQos qos - a `DomainParticipantQos` for the new `DomainParticipant`. When this set of `QosPolicy` settings is inconsistent, no `DomainParticipant` is created.

in *DomainParticipantListener a_listener* - a reference to the
DomainParticipantListener instance which will be attached to the new
DomainParticipant. It is permitted to use null as the value of the listener:
this behaves as a DomainParticipantListener whose operations perform
no action.

in *int mask* - a bit-mask in which each bit enables the invocation of the
DomainParticipantListener for a certain status.

Return Value

DomainParticipant - Return value is a reference to the newly created
DomainParticipant. In case of an error, the null reference is returned.

Detailed Description

This operation creates a new DomainParticipant, with the desired
DomainParticipantQos and attaches the optionally specified
DomainParticipantListener to it. The DomainParticipant signifies that
the calling application intends to join the Domain identified by the domainId
argument.

If the specified QosPolicy settings are not consistent, the operation will fail; no
DomainParticipant is created and the operation returns the null reference. To
delete the DomainParticipant the operation delete_participant must be
used.

Identifying the Domain

The DomainParticipant will attach to the Domain that is specified by the
domainId parameter. This parameter consists of an integer specified in the Id tag
in the configuration file. Note that to make multiple connections to a Domain (create
multiple Participants for the same Domain) within a single process, all of the
Participants must use the same identification (*i.e.* all use the same domain Id).

The constant DOMAIN_ID_DEFAULT can be used for this parameter. If this is done
the value of Id tag from the configuration file specified by the environment variable
called OSPL_URI will be used.

It is recommended to use this domain Id in conjunction with the OSPL_URI
environment variable instead of hard-coding a domain Id into your application,
since this gives you much more flexibility in the deployment phase of your product.
See also Section 1.3.2.1, *The OSPL_URI environment variable*, in the Deployment
Guide.

Default QoS

The constant `PARTICIPANT_QOS_DEFAULT` can be used as parameter `qos` to create a `DomainParticipant` with the default `DomainParticipantQos` as set in the `DomainParticipantfactory`. The effect of using `PARTICIPANT_QOS_DEFAULT` is the same as calling the operation `get_default_participant_qos` and using the resulting `DomainParticipantQos` to create the `DomainParticipant`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `DomainParticipantListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `DomainParticipantListener`:

- `INCONSISTENT_TOPIC_STATUS`   *(propagated)*
- `OFFERED_DEADLINE_MISSED_STATUS`   *(propagated)*
- `REQUESTED_DEADLINE_MISSED_STATUS`   *(propagated)*
- `OFFERED_INCOMPATIBLE_QOS_STATUS`   *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`   *(propagated)*
- `SAMPLE_LOST_STATUS`   *(propagated)*
- `SAMPLE_REJECTED_STATUS`   *(propagated)*
- `DATA_ON_READERS_STATUS`   *(propagated)*
- `DATA_AVAILABLE_STATUS`   *(propagated)*
- `LIVELINESS_LOST_STATUS`   *(propagated)*
- `LIVELINESS_CHANGED_STATUS`   *(propagated)*
- `PUBLICATION_MATCHED_STATUS`   *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS`   *(propagated).*

⚠ Be aware that the `PUBLICATION_MATCHED_STATUS` and `SUBSCRIPTION_MATCHED_STATUS` are not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `Listener` of a contained entity, the `Listener` on that contained entity is invoked instead of the `DomainParticipantListener`. This means that a status change on a contained entity only invokes the `DomainParticipantListener` if the contained entity itself does not handle the trigger event generated by the status change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are "Read Communication Statuses" and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a DataReader, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipant Listener` for the `DATA_AVAILABLE_STATUS` (in that order).

3.2.2.2 delete_participant

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
public int
    delete_participant
        (DomainParticipant a_participant);
```

Description

This operation deletes a `DomainParticipant`.

Parameters

in DomainParticipant a_participant - a reference to the DomainParticipant, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_OUT_OF_RESOURCES or RETCODE_ PRECONDITION_NOT_MET.

Detailed Description

This operation deletes a DomainParticipant. A DomainParticipant cannot be deleted when it has any attached Entity objects. When the operation is called on a DomainParticipant with existing Entity objects, the operation returns RETCODE_PRECONDITION_NOT_MET.

<u>*Return Code*</u>

When the operation returns;

- *RETCODE_OK* - the DomainParticipant is deleted

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter a_participant is not a valid DomainParticipant reference.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - the DomainParticipant contains one or more Entity objects.

3.2.2.3 **get_default_participant_qos**

Scope

DDS.DomainParticipantFactory

Synopsis

```
import DDS.*;
public int
   get_default_participant_qos
      (DomainParticipantQosHolder qos);
```

Description

This operation gets the default DomainParticipantQos of the DomainParticipantFactory.

Parameters

inout DomainParticipantQosHolder qos - a reference to the destination `DomainParticipantQosHolder` object in which the default `DomainParticipantQos` for the `DomainParticipantFactory` is written.

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets the default `DomainParticipantQos` of the `DomainParticipantFactory` (that is the object with the `QosPolicy` settings) which is used for newly created `DomainParticipant` objects, in case the constant `PARTICIPANT_QOS_DEFAULT` is used. The default `DomainParticipantQos` is only used when the constant is supplied as parameter `qos` to specify the `DomainParticipantQos` in the `create_participant` operation. The application must provide an object of the `DomainParticipantQos` class in which the `QosPolicy` settings can be stored and provide a reference to the object. The operation writes the default `QosPolicy` settings to the object referenced to by `qos`. Any settings in the object are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_participant_qos`, or, if the call was never made, the default `QosPolicy` values as defined in Table 2, *DCPS Holder Classes*, on page 12.

Return Code

When the operation returns:

• *RETCODE_OK* - the default `DomainParticipant QosPolicy` settings of this `DomainParticipantFactory` have successfully been copied into the specified `DomainParticipantQosHolder` parameter.

• *RETCODE_ERROR* - an internal error has occurred.

• *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.4 get_instance

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
```

```
public static DomainParticipantFactory
   get_instance
      (void);
```

Description

This operation returns the `DomainParticipantFactory` singleton.

Parameters

<none>

Return Value

DomainParticipantFactory - return value is a reference to the `DomainParticipantFactory`.

Detailed Description

This operation returns the `DomainParticipantFactory` singleton. The operation is idempotent, that is, it can be called multiple times without side-effects and it returns the same `DomainParticipantFactory` instance.

The operation is static and must be called upon its class (`DomainParticipantFactory.get_instance`).

The pre-defined value `TheParticipantFactory` can also be used as an alias for the singleton factory returned by the operation `get_instance`.

3.2.2.5 **get_qos**

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
public int
   get_qos
      (DomainParticipantFactoryQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a `DomainParticipantFactory`.

Parameters

inout DomainParticipantFactoryQosHolder qos - a reference to the destination `DomainparticipantFactoryQosHolder` object in which the `QosPolicy` settings will be copied.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation allows access to the existing set of QoS policies of a
DomainParticipantFactory on which this operation is used. This
DomainparticipantFactoryQos is stored at the location pointed to by the qos
parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this
  DomainParticipantFactory has successfully been copied into the specified
  DomainParticipantFactoryQosHolder parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.2.2.6 lookup_participant

Scope

DDS.DomainParticipantFactory

Synopsis

```
import DDS.*;
public DomainParticipant
   lookup_participant
      (int domainId);
```

Description

This operation retrieves a previously created DomainParticipant belonging to
the specified domainId.

Parameters

in int domainId - the ID of the Domain for which a joining
DomainParticipant should be retrieved. This should be the ID as specified
in the configuration file.

Return Value

DomainParticipant - Return value is a reference to the `DomainParticipant`
  retrieved. When no such `DomainParticipant` is found, the `null` reference is
  returned.

Detailed Description

This operation retrieves a previously created `DomainParticipant` belonging to
the specified `domainId`. If no such `DomainParticipant` exists, the operation will
return `null`. The `domainId` used to search for a specific `DomainParticipant`
must be identical to the `domainId` that was used to create that specific
`DomainParticipant`.

If multiple `DomainParticipant` entities belonging to the specified `domainId`
exist, then the operation will return one of them. It is not specified which one. See
also Section 3.2.2.1, *create_participant*, on page 190.

3.2.2.7 set_default_participant_qos

Scope

    DDS.DomainParticipantFactory

Synopsis

    import DDS.*;
        int
            set_default_participant_qos
            (DomainParticipantQos qos);

Description

This operation sets the default `DomainParticipantQos` of the
`DomainParticipantFactory`.

Parameters

in DomainParticipantQos qos - an object of the `DomainParticipantQos`
  class, which contains the new default `DomainParticipantQos` for the newly
  created `DomainParticipants`.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation sets the default `DomainParticipantQos` of the `DomainParticipantFactory` (that is the object with the `QosPolicy` settings) which is used for newly created `DomainParticipant` objects, in case the constant `PARTICIPANT_QOS_DEFAULT` is used. The default `DomainParticipantQos` is only used when the constant is supplied as parameter `qos` to specify the `DomainParticipantQos` in the `create_participant` operation. The `DomainParticipantQos` is always self consistent, because its policies do not depend on each other. This means that this operation never returns the `RETCODE_INCONSISTENT_POLICY`.

The values set by this operation are returned by `get_default_participant_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `DomainParticipantQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.8 **set_qos**

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
public int
   set_qos
      (DomainParticipantFactoryQos qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipantFactory`.

Parameters

in DomainParticipantFactoryQos qos - must contain the new set of `QosPolicy` settings for the `DomainParticipantFactory`.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipantFactory`. The parameter qos must contain the object with the `QosPolicy` settings.

The set of `QosPolicy` settings specified by the qos parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided the operation returned `RETCODE_OK`).

Return Code

When the operation returns:

- *RETCODE_OK* - the new `DomainParticipantFactoryQos` is set.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.9 **delete_domain**

Scope

DDS.DomainParticipantFactory

Synopsis

```
import DDS.*;
public int
   delete_domain
      (Domain a_domain);
```

Description

This operation deletes a `Domain` proxy.

Parameters

in Domain a_domain - a pointer to the `Domain` proxy, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER or
RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation deletes a `Domain` proxy.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Domain` proxy is deleted.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `a_domain` is not a valid `Domain` proxy.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.10 **lookup_domain**

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
public Domain
   lookup_domain
      (int domainId);
```

Description

This operation retrieves a previously created `Domain` proxy belonging to the specified `domainId` or creates a new `Domain` proxy if no `Domain` proxy yet exists but the `Domain` itself is available.

Parameters

in int domainId - the ID of the Domain for which a Domain proxy should be retrieved. This should be the ID as specified in the configuration file.

Return Value

Domain - Return value is a pointer to the `Domain` proxy retrieved. When no such `Domain` proxy is found or could be created, the `null` reference is returned.

Detailed Description

This operation retrieves a previously created Domain proxy belonging to the specified `domainId` or creates a new Domain proxy if no Domain proxy was found, but the `DomainId` does refer to a valid Domain. If no such Domain exists or could be created, the operation will return `null`. See also Section 3.2.2.1, *create_participant*, on page 190.

3.2.2.11 **delete_contained_entities**

Scope

```
DDS.DomainParticipantFactory
```

Synopsis

```
import DDS.*;
    public int
        delete_contained_entities
            (void);
```

Description

This operation deletes all of the `Entity` objects that were created on the `DomainParticipantFactory`.

Parameters

<none>

Return Value

`int` - Possible return codes of the operation are:
  `RETCODE_OK, RETCODE_ERROR, RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes all of the `Entity` objects that were created on the `DomainParticipantFactory` (it deletes all contained `DomainParticipant` objects). Prior to deleting each contained `Entity`, this operation regressively calls the `delete_contained_entities` operation on each `Participant`. In other words, this operation cleans up *all* `Entity` objects in the process.

NOTE: The operation will return `PRECONDITION_NOT_MET` if the any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained `DataReader` cannot be deleted because the application has called a `read` or `take` operation and has not called the corresponding

`return_loan` operation to return the loaned samples. In such cases, the operation does not roll back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

RETCODE_OK - all contained Entity objects are deleted.

RETCODE_ERROR - an internal error has occurred.

RETCODE_OUT_OF_RESOURCES - the Data Distribution Service ran out of resources to complete this operation.

RETCODE_PRECONDITION_NOT_MET - one or more of the contained entities are in a state where they cannot be deleted.

3.2.2.12 detach_all_domains

Scope

`DDS.DomainParticipantFactory`

Synopsis

```
import DDS.*;
public int
    detach_all_domains
        (boolean block_operations,
        boolean delete_entities);
```

Description

This operation will safely detach the application from all domains it is currently participating in.

Parameters

in boolean block_operations –
Indicates whether the application wants any operations that are called while detaching to be blocked or not.

in boolean delete_entities –
Indicates whether the application DDS entities in the 'connected' domains must be deleted synchronously during detaching.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK

Detailed Description

This operation safely detaches the application from all domains it is currently participating in. When this operation has been performed successfully, the application is no longer connected to any Domain.

- For *Federated* domains, finishing this operation successfully means that all shared memory segments have been safely un-mapped from the application process.

- For *SingleProcess* mode domains, this means that all services for all domains have been stopped. This allows graceful termination of the OSPL services that run as threads within the application. Graceful termination of services in this mode would for instance allow durability flushing of persistent data and networking termination announcement over the network.

When this call returns, further access to all domains will be denied and it will not be possible for the application to open or re-open any DDS domain.

The behavior of the `detach_all_domains` operation is determined by the *`block_operations`* and *`delete_entities`* parameters.

`block_operations` – This parameter specifies whether the application wants any DDS operation to be blocked or not while detaching. When `TRUE`, any DDS operation called during this operation will be blocked and remain blocked forever (so also after the detach operation has completed and returns to the caller). When `FALSE`, any DDS operation called during this operation may return `RETCODE_ALREADY_DELETED`.
Please note that a listener callback is *not* considered an operation in progress. Of course, if a DDS operation is called from *within* the listener callback, that operation *will* be blocked during the detaching if this attribute is set to `TRUE`.

`delete_entities` – This parameter specifies whether the application wants the DDS entities created by the application to be deleted (synchronously) while detaching from the domain or not. If `TRUE`, all application entities are guaranteed to be deleted when the call returns. If `FALSE`, application entities will not explicitly be deleted by this operation.
In *federated* mode, the splice-daemon will delete them asynchronously after this operation has returned successfully. In *SingleProcess* mode this attribute is ignored and clean up will always be performed, as this cannot be delegated to a different process.

⚠ **NOTE**: In *federated* mode when the `detach_all_domain` operation is called with *`block_operations`* set to `FALSE` and *`delete_entities`* also `FALSE` then the DDS operations which are in progress and which are waiting for some condition to become true (or waiting for an event to occur) while the detach operation is performed *may* be blocked.

When the operation returns:

- *RETCODE_OK* – the application is detached from all domains.

3.2.3 Class Domain

The purpose of this class is to represent the `Domain` and allow certain `Domain`-wide operations to be performed. In essence it is a proxy to the `Domain`.

A `Domain` is a distributed concept that links all the applications that must be able to communicate with each other. It represents a communication plane: only the `Publishers` and the `Subscribers` attached to the same `Domain` can interact.

This class currently implements one function:

- It allows for a snapshot to be taken of all persistent data available within this `Domain` on local node level.

The interface description of this class is as follows:

```
/*
 * interface Domain
 */
public class Domain {
   public int
      create_persistent_snapshot(
         String partition_expression,
         String topic_expression,
         String URI);
};
```

The following sections describe the usage of all `Domain` operations.

3.2.3.1 create_persistent_snapshot

Scope

```
DDS.Domain
```

Synopsis

```
public int
   create_persistent_snapshot(
      String partition_expression,
      String topic_expression,
      String URI);
```

Description

This operation will create a snapshot of all persistent data matching the provided partition and topic expressions and store the snapshot at the location indicated by the URI. Only persistent data available on the local node is considered.

Parameters

in `String partition_expression` - The expression of all partitions involved in the snapshot; this may contain wildcards.

in `String topic_expression` - The expression of all topics involved in the snapshot; this may contain wildcards.

in `String uri` - The location where to store the snapshot. Currently only directories are supported.

<u>*Return Value*</u>

`int` - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation will create a snapshot of all persistent data matching the provided partition and topic expressions and store the snapshot at the location indicated by the URI. Only persistent data available on the local node is considered. This operation will fire an event to trigger the snapshot creation by the durability service and then return while the durability service fulfills the snapshot request; if no durability service is available then there is no persistent data available and the operation will return `OK` as a snapshot of an empty store is an empty store.

The created snapshot can then be used as the persistent store for the durability service next time it starts up by configuring the location of the snapshot as the persistent store in the configuration file. The durability service will then use the snapshot as the regular store (and can thus also alter its contents).

<u>*Return Code*</u>

When the operation returns:

- *RETCODE_OK* – The persistent snapshot is (being) created.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `partition_expression`, `topic_expression` or `uri` is a `null` reference.
- *RETCODE_ALREADY_DELETED* - the `Domain` proxy has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.4 **DomainParticipantListener Interface**

Since a `DomainParticipant` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `DomainParticipantListener`. This interface must be implemented by the application. A user-defined class must be provided by the application which must extend from the `DomainParticipantListener` class. All `DomainParticipantListener` operations must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

⚠ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The `DomainParticipantListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `DomainParticipantListener` is related to changes in communication status `StatusConditions`.

The interface description of this class is as follows:

```
 public interface DomainParticipantListener
 {
 //
 // extends interface TopicListener
 //
 // void
 //     on_inconsistent_topic
 //         (Topic the_topic,
 //           InconsistentTopicStatus status);
 //
 // extends interface PublisherListener
 //
 // void
 //     on_offered_deadline_missed
 //         (DataWriter writer,
 //           OfferedDeadlineMissedStatus status);

 // void
 //     on_offered_incompatible_qos
 //         (DataWriter writer,
 //           OfferedIncompatibleQosStatus status);

 // void
 //     on_liveliness_lost
 //         (DataWriter writer,
 //           LivelinessLostStatus status);
```

```
//  void
//      on_publication_matched
//          (DataWriter writer,
//            PublicationMatchedStatus status);
//
// extends interface SubscriberListener
//
// void
//      on_data_on_readers
//          (Subscriber subs);
// void
//      on_requested_deadline_missed
//          (DataReader reader,
//            RequestedDeadlineMissedStatus status);

// void
//      on_requested_incompatible_qos
//          (DataReader reader,
//            RequestedIncompatibleQosStatus status);

// void
//      on_sample_rejected
//          (DataReader reader,
//            SampleRejectedStatus status);

// void
//      on_liveliness_changed
//          (DataReader reader,
//            LivelinessChangedStatus status);

// void
//      on_data_available
//          (DataReader reader);

// void
//      on_subscription_matched
//          (DataReader reader,
//            SubscriptionMatchedStatus status);

// void
//      on_sample_lost
//          (DataReader reader,
//            SampleLostStatus status);
//
// implemented API operations
//        <no operations>
//
};
```

PRISMTECH

The following paragraphs list all `DomainParticipantListener` operations. Since these operations are all inherited, they are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.2.4.1 on_data_available (inherited, abstract)

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_data_available
      (DataReader reader);
```

3.2.4.2 on_data_on_readers (inherited, abstract)

This operation is inherited and therefore not described here. See the interface `SubscriberListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_data_on_readers
      (Subscriber subs);
```

3.2.4.3 on_inconsistent_topic (inherited, abstract)

This operation is inherited and therefore not described here. See the interface `TopicListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_inconsistent_topic
      (Topic the_topic,
        InconsistentTopicStatus status);
```

3.2.4.4 on_liveliness_changed (inherited, abstract)

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_liveliness_changed
```

```
(DataReader reader,
  LivelinessChangedStatus status);
```

3.2.4.5 **on_liveliness_lost (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
  on_liveliness_lost
    (DataWriter writer,
      LivelinessLostStatus status);
```

3.2.4.6 **on_offered_deadline_missed (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
  on_offered_deadline_missed
    (DataWriter writer,
      OfferedDeadlineMissedStatus status);
```

3.2.4.7 **on_offered_incompatible_qos (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
  on_offered_incompatible_qos
    (DataWriter writer,
      OfferedIncompatibleQosStatus status);
```

3.2.4.8 **on_publication_matched (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
  on_publication_matched
    (DataWriter writer,
```

```
                             PublicationMatchedStatus status);
```

3.2.4.9 **on_requested_deadline_missed (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_requested_deadline_missed
      (DataReader reader,
         RequestedDeadlineMissedStatus status);
```

3.2.4.10 **on_requested_incompatible_qos (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_requested_incompatible_qos
      (DataReader reader,
         RequestedIncompatibleQosStatus status);
```

3.2.4.11 **on_sample_lost (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_sample_lost
      (DataReader reader,
         SampleLostStatus status);
```

3.2.4.12 **on_sample_rejected (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_sample_rejected
      (DataReader reader,
         SampleRejectedStatus status);
```

3.2.4.13 **on_subscription_matched (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_subscription_matched
      (DataReader reader,
          SubscriptionMatchedStatus status);
```

3.2.5 **ExtDomainParticipantListener interface**

The `ExtDompainParticipantListener` interface is a subtype of both `DomainParticipantListener` and `ExtTopicListener` and thereby provides an additional OpenSplice-specific callback, `on_all_disposed_data`, usable from the `DomainParticipant`.

⚠️ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The interface description of this class is as follows:

```
public interface ExtDomainParticipantListener extends
ExtTopicListener, DomainParticiantListener
{
//
// extends interface ExtTopicListener
//
// void
//    on_all_data_disposed
//        (Topic the_topic);
//
// extends interface TopicListener
//
// void
//    on_inconsistent_topic
//        (Topic the_topic,
//          InconsistentTopicStatus status);
//
// extends interface PublisherListener
//
// void
//    on_offered_deadline_missed
//        (DataWriter writer,
//          OfferedDeadlineMissedStatus status);
// void
//    on_offered_incompatible_qos
```

```
//          (DataWriter writer,
//           OfferedIncompatibleQosStatus status);
// void
//     on_liveliness_lost
//         (DataWriter writer,
//           LivelinessLostStatus status);
// void
//     on_publication_matched
//         (DataWriter writer,
//           PublicationMatchedStatus status);
//
// extends interface SubscriberListener
//
// void
//     on_data_on_readers
//         (Subscriber subs);
// void
//     on_requested_deadline_missed
//         (DataReader reader,
//           RequestedDeadlineMissedStatus status);
// void
//     on_requested_incompatible_qos
//         (DataReader reader,
//           RequestedIncompatibleQosStatus status);
// void
//     on_sample_rejected
//         (DataReader reader,
//           SampleRejectedStatus status);
// void
//     on_liveliness_changed
//         (DataReader reader,
//           LivelinessChangedStatus status);
// void
//     on_data_available
//         (DataReader reader);
// void
//     on_subscription_matched
//         (DataReader reader,
//           SubscriptionMatchedStatus status);
// void
//     on_sample_lost
//         (DataReader reader,
//           SampleLostStatus status);
//
// implemented API operations
// <no operations>
//
};
```

The following paragraphs list all `ExtDomainParticipantListener` operations. Since these operations are all inherited, they are listed but not fully described because they are not implemented in this class. The full descriptions of these operations are given in the classes from which they are inherited.

3.2.5.1 **on_all_data_disposed (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `ExtTopicListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_all_data_disposed
      (DDS.Topic the_topic);
```

3.2.5.2 **on_data_available (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_data_available
      (DDS.DataReader reader);
```

3.2.5.3 **on_data_on_readers (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `SubscriberListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_data_on_readers
      (DDS.Subscriber subs);
```

3.2.5.4 **on_inconsistent_topic (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `TopicListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_inconsistent_topic
      (DDS.Topic the_topic,
```

```
                                    DDS.InconsistentTopicStatus status);
```

3.2.5.5 on_liveliness_changed (inherited, abstract)

This operation is inherited and therefore not described here. See the interface
`DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_liveliness_changed
      (DDS.DataReader reader,
         DDS.LivelinessChangedStatus status);
```

3.2.5.6 on_liveliness_lost (inherited, abstract)

This operation is inherited and therefore not described here. See the interface
`DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_liveliness_lost
      (DDS.DataWriter writer,
         DDS.LivelinessLostStatus status);
```

3.2.5.7 on_offered_deadline_missed (inherited, abstract)

This operation is inherited and therefore not described here. See the interface
`DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_offered_deadline_missed
      (DDS.DataWriter writer,
         DDS.OfferedDeadlineMissedStatus status);
```

3.2.5.8 on_offered_incompatible_qos (inherited, abstract)

This operation is inherited and therefore not described here. See the interface
`DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_offered_incompatible_qos
      (DDS.DataWriter writer,
         DDS.OfferedIncompatibleQosStatus status);
```

3.2.5.9 **on_publication_matched (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_publication_matched
      (DDS.DataWriter writer,
         DDS.PublicationMatchedStatus status);
```

3.2.5.10 **on_requested_deadline_missed (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_requested_deadline_missed
      (DDS.DataReader reader,
         DDS.RequestedDeadlineMissedStatus status);
```

3.2.5.11 **on_requested_incompatible_qos (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_requested_incompatible_qos
      (DDS.DataReader reader,
         DDS.RequestedIncompatibleQosStatus status);
```

3.2.5.12 **on_sample_lost (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_sample_lost
      (DDS.DataReader reader,
         DDS.SampleLostStatus status);
```

3.2.5.13 **on_sample_rejected (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_sample_rejected
      (DDS.DataReader reader,
        DDS.SampleRejectedStatus status);
```

3.2.5.14 **on_subscription_matched (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
void
   on_subscription_matched
      (DDS.DataReader reader,
        DDS.SubscriptionMatchedStatus status);
```

3.3 **Topic-Definition Module**
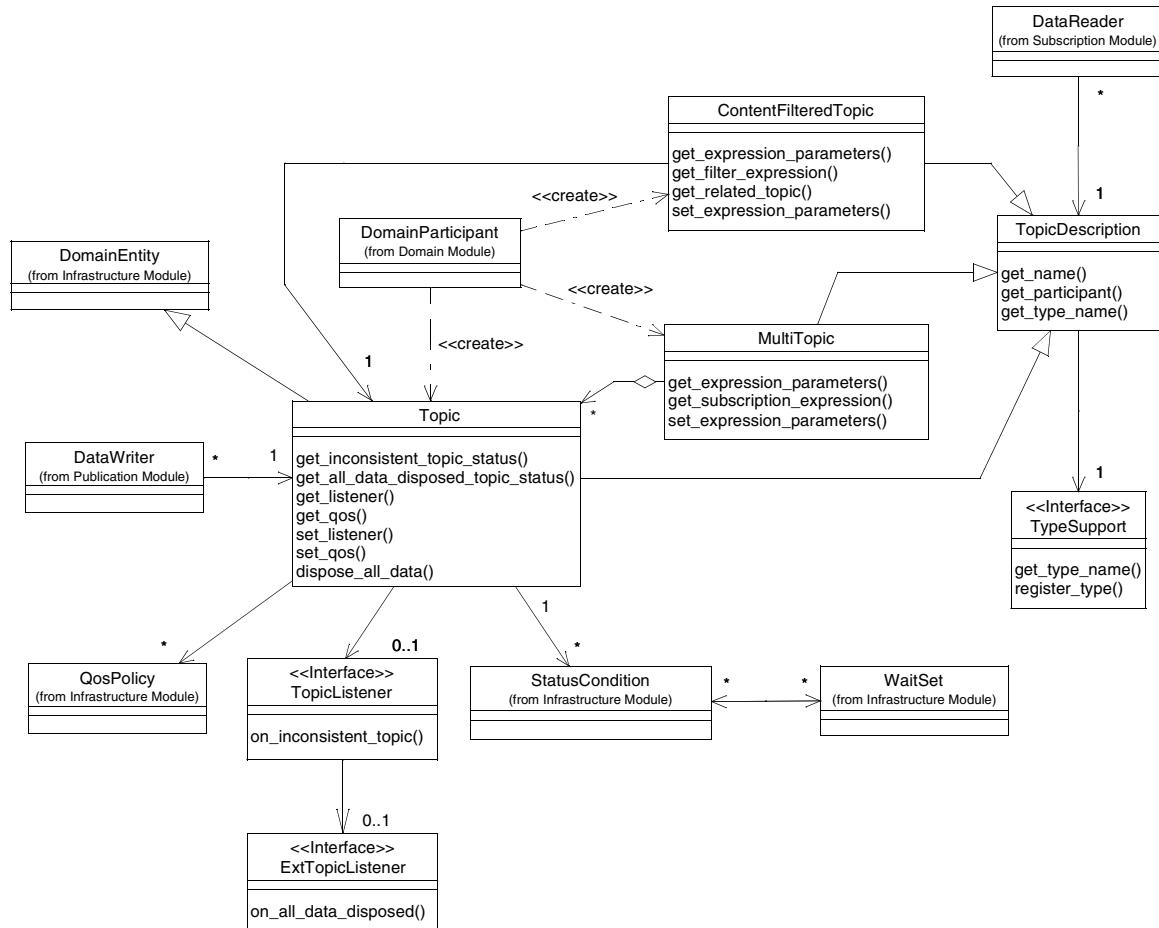
Figure 16 DCPS Topic-Definition Module's Class Model

This module contains the following classes:

- `TopicDescription` (abstract)
- `Topic`
- `ContentFilteredTopic`
- `MultiTopic`
- `TopicListener` (interface)
- Topic-Definition type specific classes

PRISMTECH

"Topic-Definition type specific classes" contains the generic class and the generated data type specific classes. For each data type, a data type specific class `<type>TypeSupport` is generated (based on IDL) by calling the pre-processor
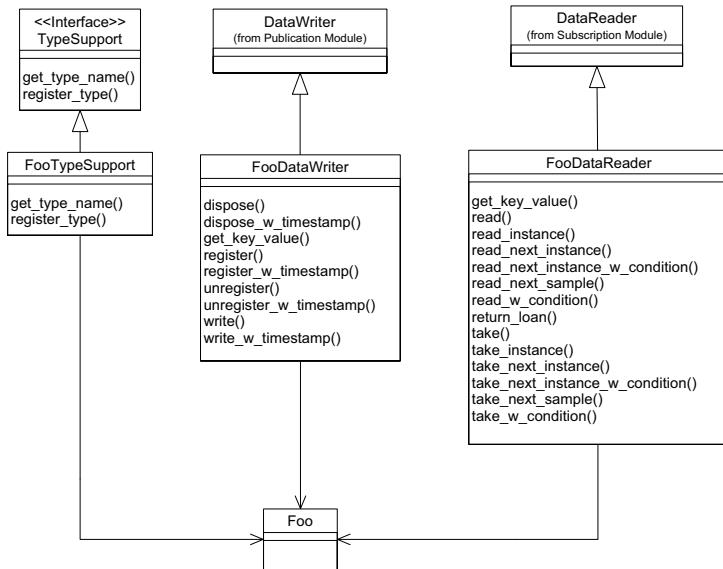
Figure 17 Pre-processor Generation of the Typed Classes for Data Type "Foo"

For instance, for the fictional data type `Foo` (this also applies to other types); "Topic-Definition type specific classes" contains the following classes:

- `TypeSupport` (abstract)
- `FooTypeSupport`

`Topic` objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A `Topic` is meant to fulfil that purpose: it associates a name (unique in the `Domain`), a data type, and `TopicQos` related to the data itself.

3.3.1 Interface TopicDescription

This is the interface for `Topic`, `ContentFilteredTopic` and `MultiTopic`.

The `TopicDescription` attribute `type_name` defines an unique data type that is made available to the Data Distribution Service via the `TypeSupport`. `TopicDescription` has also a `name` that allows it to be retrieved locally.

The interface description is as follows:

```
public interface TopicDescription
{
//
```

```
              // implemented API operations
              //
                 public String
                    get_type_name
                       (void);
                 public String
                    get_name
                       (void);
                 public DomainParticipant
                    get_participant
                       (void);
              };
```

The following paragraphs describe the usage of all `TopicDescription` operations.

3.3.1.1 get_name

Scope

```
    DDS.TopicDescription
```

Synopsis

```
    import DDS.*;
    public String
       get_name
          (void);
```

Description

This operation returns the name used to create the `TopicDescription`.

Parameters

<none>

Return Value

String - return value is the name of the `TopicDescription`.

Detailed Description

This operation returns the name used to create the `TopicDescription`.

3.3.1.2 get_participant

Scope

```
    DDS.TopicDescription
```

Synopsis

```
    import DDS.*;
```

```
public DomainParticipant
   get_participant
      (void);
```

Description

This operation returns the `DomainParticipant` associated with the `TopicDescription` or the `null` reference.

Parameters

<none>

Return Value

`DomainParticipant` - a reference to the `DomainParticipant` associated with the `TopicDescription` or the `null` reference.

Detailed Description

This operation returns the `DomainParticipant` associated with the `TopicDescription`. Note that there is exactly one `DomainParticipant` associated with each `TopicDescription`. When the `TopicDescription` was already deleted (there is no associated `DomainParticipant` any more), the `null` reference is returned.

3.3.1.3 get_type_name

Scope

```
DDS.TopicDescription
```

Synopsis

```
import DDS.*;
public String
   get_type_name
      (void);
```

Description

This operation returns the registered name of the data type associated with the `TopicDescription`.

Parameters

<none>

Return Value

`String` - return value is the name of the data type of the `TopicDescription`.

Detailed Description

This operation returns the registered name of the data type associated with the `TopicDescription`.

3.3.2 Interface Topic

`Topic` is the most basic description of the data to be published and subscribed.

A `Topic` is identified by its `name`, which must be unique in the whole `Domain`. In addition (by virtue of extending `TopicDescription`) it fully identifies the type of data that can be communicated when publishing or subscribing to the `Topic`.

`Topic` is the only `TopicDescription` that can be used for publications and therefore a specialized `DataWriter` is associated to the `Topic`.

The interface description is as follows:

```
 public interface Topic
{
//
// extends interface Entity
//
// public StatusCondition
//      get_statuscondition
//          (void);
// public int
//      get_status_changes
//          (void);
// public int
//      enable
//          (void);
//
// extends interface TopicDescription
//
// public String
//      get_type_name
//          (void);

// public String
//      get_name
//          (void);

// public DomainParticipant
//      get_participant
//          (void);
//
// implemented API operations
//
    public int
       set_qos
          (TopicQos qos);
```

PRISMTECH

```
                    public int
                       get_qos
                          (TopicQosHolder qos);
                    public int
                       set_listener
                          (TopicListener a_listener,
                          int mask);
                    public TopicListener
                       get_listener
                          (void);
                    public int
                       get_inconsistent_topic_status
                          (InconsistentTopicStatusHolder status);
                    public int
                       get_all_data_disposed_topic_status
                          (AllDataDisposedTopicStatusHolder status);
                    public int dispose_all_data ();
                 };
```

The following paragraphs describe the usage of all `Topic` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.3.2.1 enable (inherited)

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   enable
      (void);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.3.2.2 get_inconsistent_topic_status

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public int
   get_inconsistent_topic_status
      (InconsistentTopicStatusHolder status);
```

Description

This operation obtains the `InconsistentTopicStatus` of the `Topic`.

Parameters

inout InconsistentTopicStatusHolder status - the contents of the
   `InconsistentTopicStatus` object of the `Topic` will be copied into the
   `InconsistentTopicStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or
`RETCODE_OUT_OF_RESOURCES.`

Detailed Description

This operation obtains the `InconsistentTopicStatus` of the `Topic`. The
`InconsistentTopicStatus` can also be monitored using a `TopicListener` or
by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `InconsistentTopicStatus` of this `Topic` has
  successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Topic` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.3.2.3 **get_all_data_disposed_topic_status**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public int
   get_all_data_disposed_topic_status
      (AllDataDisposedTopicStatusHolder status);
```

Description

This operation obtains the `AllDataDisposedTopicStatus` of the `Topic`.

PRISMTECH

Parameters

inout AllDataDisposedTopicStatusHolder status - the contents of the AllDataDisposedTopicStatus object of the `Topic` will be copied into the AllDataDisposedTopicStatusHolder specified by status.

Return Value

int – Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES.`

Detailed Description

This operation obtains the AllDataDisposedTopicStatus of the Topic. The AllDataDisposedTopicStatus can also be monitored using a ExtTopicListener or by using the associated StatusCondition.

Return Code

When the operation returns:

- *RETCODE_OK* – the current `InconsistentTopicStatus` of this `Topic` has successfully been copied into the specified a_status parameter.

- *RETCODE_ERROR* – an internal error has occurred.

- *RETCODE_ALREADY_DELETED* – the `Topic` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* – the Data Distribution Service ran out of resources to complete this operation.

3.3.2.4 dispose_all_data

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public int
   dispose_all_data ();
```

Description

This operation allows the application to dispose of all of the instances for a particular topic without the network overhead of using a separate `dispose` call for each instance.

Parameters

Return Value

`int` - Possible return codes of the operation are:
    `RETCODE_OK, RETCODE_ERROR, RETCODE_OUT_OF_RESOURCES,`
    `RETCODE_ALREADY_DELETED, RETCODE_NOT_ENABLED.`

DetailedDescription

This operation allows the application to dispose of all of the instances for a particular topic without the network overhead of using a separate `dispose` call for each instance. Its effect is equivalent to invoking a separate `dispose` operation for each individual instance on the `DataWriter` that owns it. (See the description of `FooDataWriter.dispose` in Section 3.4.2.33, *dispose*, on page 305.)

(The `dispose_all_data` is an asynchronous C&M operation that is not part of a coherent update; it operates on the DataReaders history cache and not on the incomplete transactions. The `dispose_all_data` is effectuated as soon as a transaction becomes complete and is inserted into the DataReaders history cache; at that point messages will be inserted according to the `destination_order` qos policy. For `BY_SOURCE_TIMESTAMP` all messages older than the `dispose_all_data` will be disposed and all newer will be alive; for `BY_RECEPTION_TIMESTAMP` all messages will be alive if the transaction is completed after receiving the `dispose_all_data` command.)

⚠️ This operation *only* sets the instance state of the instances concerned to `NOT_ALIVE_DISPOSED`. It does *not* unregister the instances, and so does not automatically clean up the memory that is claimed by the instances in both the `DataReaders` and `DataWriters`.

Blocking

The blocking (or nonblocking) behaviour of this call is undefined.

Concurrency

If there are subsequent calls to this function before the action has been completed (completion of the disposes on all nodes, not simply return from the function), then the behaviour is undefined.

Other notes

The effect of this call on `disposed_generation_count`, `generation_rank` and `absolute_generation_rank` is undefined.

Return Code

• *RETCODE_OK* - a request to dispose the topic has been sucessfully queued.

• *RETCODE_ERROR* - and internal error has occured.

PRISMTECH

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_ALREADY_DELETED* - the Topic has already been deleted.
- *RETCODE_NOT_ENABLED* - the Topic is not enabled.

3.3.2.5 get_listener

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public TopicListener
    get_listener
        (void);
```

Description

This operation allows access to a `TopicListener`.

Parameters

<none>

Return Value

TopicListener - result is a reference to the `TopicListener` attached to the Topic.

Detailed Description

This operation allows access to a `TopicListener` attached to the `Topic`. When no `TopicListener` was attached to the `Topic`, the `null` reference is returned.

3.3.2.6 get_name (inherited)

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public String
    get_name
        (void);
```

3.3.2.7 **get_participant (inherited)**

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public DomainParticipant
   get_participant
      (void);
```

3.3.2.8 **get_qos**

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public int
   get_qos
      (TopicQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a `Topic`.

Parameters

inout TopicQosHolder qos - a reference to the destination `TopicQosHolder` object in which the `QosPolicy` settings will be copied.

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `Topic` on which this operation is used. This `TopicQos` is stored at the location referenced by the `qos` parameter.

Return Code

When the operation returns:

• *RETCODE_OK* - the existing set of QoS policy values applied to this `Topic` has successfully been copied into the specified `TopicQosHolder` parameter.

- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the Topic has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.3.2.9 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the interface Entity for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
      (void);
```

3.3.2.10 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the interface Entity for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
      (void);
```

3.3.2.11 get_type_name (inherited)

This operation is inherited and therefore not described here. See the interface TopicDescription for further explanation.

Synopsis

```
import DDS.*;
public String
   get_type_name
      (void);
```

3.3.2.12 set_listener

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public int
   set_listener
```

```
(TopicListener a_listener,
  int mask);
```

Description

This operation attaches a `TopicListener` to the `Topic`.

Parameters

in TopicListener a_listener - a reference to the `TopicListener` instance, which will be attached to the `Topic`.

in int mask - a bit mask in which each bit enables the invocation of the `TopicListener` for a certain status.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation attaches a `TopicListener` to the `Topic`. Only one `TopicListener` can be attached to each `Topic`. If a `TopicListener` was already attached, the operation will replace it with the new one. When `a_listener` is the `null` reference, it represents a listener that is treated as a NOOP[1] for all statuses activated in the bitmask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that plain communication status changes. For each plain communication status activated in the `mask`, the associated `TopicListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>` from inside the listener it will see the status already reset. An exception to this rule is the `null` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `TopicListener`:

• `INCONSISTENT_TOPIC_STATUS`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available

---

1.   Short for **No-Op**eration, an instruction that peforms nothing at all.

statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

In case a communication status is not activated in the `mask` of the `TopicListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Topic` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

Return Code

When the operation returns:

- *RETCODE_OK* - the `TopicListener` is attached
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the `Topic` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.3.2.13 set_qos

Scope

```
DDS.Topic
```

Synopsis

```
import DDS.*;
public int
   set_qos
       (TopicQos qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `Topic`.

Parameters

in TopicQos qos - contains the new set of `QosPolicy` settings for the `Topic`.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,`
`RETCODE_UNSUPPORTED,     RETCODE_ALREADY_DELETED,`
`RETCODE_OUT_OF_RESOURCES, RETCODE_IMMUTABLE_POLICY` or
`RETCODE_INCONSISTENT_POLICY.`

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `Topic`. The parameter `qos` contains the object with the `QosPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QosPolicy` setting for an enabled `Topic`, which can only be set before the `Topic` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the currently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` settings (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

The set of `QosPolicy` settings specified by the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned `RETCODE_OK`).

Return Code

When the operation returns:

- *RETCODE_OK* - the new `TopicQos` is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `qos` is not a valid `TopicQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value.
- *RETCODE_UNSUPPORTED* - one or more of the selected QosPolicy values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the `Topic` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter `qos` contains an immutable `QosPolicy` setting with a different value than set during enabling of the `Topic`.
- *RETCODE_INCONSISTENT_POLICY* - the parameter `qos` contains conflicting `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.3.3 **Interface ContentFilteredTopic**

`ContentFilteredTopic` is a specialization of `TopicDescription` that allows for content based subscriptions.

`ContentFilteredTopic` describes a more sophisticated subscription that indicates the `Subscriber` does not necessarily want to see all values of each instance published under the `Topic`. Rather, it only wants to see the values whose contents satisfy certain criteria. Therefore this interface must be used to request content-based subscriptions.

The selection of the content is done using the SQL based filter with parameters to adapt the filter clause.

The interface description is as follows:

```
 public interface ContentFilteredTopic
{
//
// extends interface TopicDescription
//
// public String
//     get_type_name
//         (void);

// public String
//     get_name
//         (void);

// public DomainParticipant
//     get_participant
//         (void);
//
// implemented API operations
//
    public String
       get_filter_expression
           (void);

    public int
       get_expression_parameters
           (StringSeqHolder expression_parameters);


    public int
       set_expression_parameters
           (StringSeq expression_parameters);


    public Topic
       get_related_topic
```

```
                              (void);
        };
```

The following paragraphs describe the usage of all `ContentFilteredTopic` operations.

3.3.3.1 **get_expression_parameters**

Scope

```
DDS.ContentFilteredTopic
```

Synopsis

```
import DDS.*;
public int
   get_expression_parameters
       (StringSeqHolder expression_parameters);
```

Description

This operation obtains the expression parameters associated with the `ContentFilteredTopic`.

Parameters

inout StringSeqHolder expression_parameters - a reference to the destination `StringSeqHolder` object in which the the parameters used in the SQL expression will be copied.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES.`

Detailed Description

This operation obtains the expression parameters associated with the `ContentFilteredTopic`. That is, the parameters specified on the last successful call to `set_expression_parameters`, or if `set_expression_parameters` was never called, the parameters specified when the `ContentFilteredTopic` was created.

The resulting reference holds a sequence of strings with the parameters used in the SQL expression (i.e., the %n tokens in the expression). The number of parameters in the result sequence will exactly match the number of %n tokens in the filter expression associated with the `ContentFilteredTopic`.

<u>*Return Code*</u>

When the operation returns:

- *RETCODE_OK* - the existing set of expression parameters applied to this `ContentFilteredTopic` has successfully been copied into the specified `expression_parameters` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `ContentFilteredTopic` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.3.3.2 **get_filter_expression**

Scope

```
DDS.ContentFilteredTopic
```

Synopsis

```
import DDS.*;
public String
   get_filter_expression
      (void);
```

Description

This operation returns the `filter_expression` associated with the `ContentFilteredTopic`.

Parameters

<none>

Return Value

String - result is a handle to a string which holds the SQL filter expression.

Detailed Description

This operation returns the `filter_expression` associated with the `ContentFilteredTopic`. That is, the expression specified when the `ContentFilteredTopic` was created.

The filter expression result is a string that specifies the criteria to select the data samples of interest. It is similar to the `WHERE` clause of an SQL expression.

3.3.3.3 **get_name (inherited)**

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public String
   get_name
      (void);
```

3.3.3.4 **get_participant (inherited)**

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public DomainParticipant
   get_participant
      (void);
```

3.3.3.5 **get_related_topic**

Scope

```
DDS.ContentFilteredTopic
```

Synopsis

```
import DDS.*;
public Topic
   get_related_topic
      (void);
```

Description

This operation returns the `Topic` associated with the `ContentFilteredTopic`.

Parameters

<none>

Return Value

Topic - result is a reference to the base topic on which the filtering will be applied.

Detailed Description

This operation returns the `Topic` associated with the `ContentFilteredTopic`. That is, the `Topic` specified when the `ContentFilteredTopic` was created. This `Topic` is the base topic on which the filtering will be applied.

3.3.3.6 get_type_name (inherited)

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public String
    get_type_name
        (void);
```

3.3.3.7 set_expression_parameters

Scope

```
DDS.ContentFilteredTopic
```

Synopsis

```
import DDS.*;
public int
    set_expression_parameters
        (String[] expression_parameters);
```

Description

This operation changes the expression parameters associated with the `ContentFilteredTopic`.

Parameters

in String[] expression_parameters - a reference to a sequence of strings with the parameters used in the SQL expression (i.e., the number of %n tokens in the expression). The number of values in `expression_parameters` must be equal or greater than the highest referenced %n token in the `subscription_expression`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation changes the expression parameters associated with the `ContentFilteredTopic`. The parameter `expression_parameters` is a handle to a sequence of strings with the parameters used in the SQL expression. The number of values in `expression_parameters` must be equal or greater than the highest referenced %n token in the `filter_expression` (e.g. if `%1` and `%8` are used as parameter in the `filter_expression`, the `expression_parameters` should at least contain `n+1 = 9` values). This is the filter expression specified when the `ContentFilteredTopic` was created.

Return Code

When the operation returns:

- *RETCODE_OK* - the new expression parameters are set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the number of parameters in `expression_parameters` does not match the number of "%n" tokens in the expression for this `ContentFilteredTopic` or one of the parameters is an illegal parameter
- *RETCODE_ALREADY_DELETED* - the `ContentFilteredTopic` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.3.4 **Interface MultiTopic**

`MultiTopic` is a specialization of `TopicDescription` that allows subscriptions to combine, filter and/or rearrange data coming from several `Topics`.

`MultiTopic` allows a more sophisticated subscription that can select and combine data received from multiple `Topics` into a single data type (specified by the inherited `type_name`). The data will then be filtered (selection) and possibly re-arranged (aggregation and/or projection) according to an SQL based expression with parameters to adapt the filter clause.

The interface description is as follows:

```
 public interface MultiTopic
 {
 //
 // extends interface TopicDescription
 //
 // public String
 //     get_type_name
 //         (void);
```

```
                    // public String
                    //    get_name
                    //       (void);

                    // public DomainParticipant
                    //    get_participant
                    //       (void);
                    //
                    // implemented API operations
                    //
                       public String
                          get_subscription_expression
                             (void);

                       public int
                          get_expression_parameters
                             (StringSeqHolder expression_parameters);

                       public int
                          set_expression_parameters
                             (String[] expression_parameters);
                    };
```

The following paragraphs describe the usage of all `MultiTopic` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

NOTE: `MultiTopic` operations have not been yet been implemented. Multitopic functionality is scheduled for a future release.

3.3.4.1 get_expression_parameters

Scope

```
    DDS.MultiTopic
```

Synopsis

```
    import DDS.*;
    public int
       get_expression_parameters
          (StringSeqHolder expression_parameters);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation obtains the expression parameters associated with the `MultiTopic`.

Parameters

inout StringSeqHolder expression_parameters - a reference to the destination `StringSeqHolder` object in which the the parameters used in the SQL expression will be copied.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the expression parameters associated with the `MultiTopic`. That is, the parameters specified on the last successful call to `set_expression_parameters`, or if `set_expression_parameters` was never called, the parameters specified when the `MultiTopic` was created.

The resulting reference holds a sequence of strings with the values of the parameters used in the SQL expression (i.e., the %n tokens in the expression). The number of parameters in the result sequence will exactly match the number of %n tokens in the filter expression associated with the `MultiTopic`.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of expression parameters applied to this `MultiTopic` has successfully been copied into the specified `expression_parameters` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `MultiTopic` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.3.4.2 get_name (inherited)

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public String
   get_name
      (void);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.3.4.3 **get_participant (inherited)**

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public DomainParticipant
    get_participant
        (void);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.3.4.4 **get_subscription_expression**

Scope

```
DDS.MultiTopic
```

Synopsis

```
import DDS.*;
public String
    get_subscription_expression
        (void);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation returns the subscription expression associated with the `MultiTopic`.

Parameters

<none>

Return Value

String - result is a handle to a string which holds the SQL subscription expression.

Detailed Description

This operation returns the subscription expression associated with the `MultiTopic`. That is, the expression specified when the `MultiTopic` was created.

The subscription expression result is a string that specifies the criteria to select the data samples of interest. In other words, it identifies the selection and rearrangement of data from the associated `Topics`. It is an SQL expression where the `SELECT` clause 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.

3.3.4.5 **get_type_name (inherited)**

This operation is inherited and therefore not described here. See the interface `TopicDescription` for further explanation.

Synopsis

```
import DDS.*;
public String
   get_type_name
       (void);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.3.4.6 **set_expression_parameters**

Scope

```
DDS.MultiTopic
```

Synopsis

```
import DDS.*;
public int
   set_expression_parameters
       (String[] expression_parameters);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation changes the expression parameters associated with the `MultiTopic`.

Parameters

in String[] expression_parameters - the handle to a sequence of strings with the parameters used in the SQL expression.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation changes the expression parameters associated with the `MultiTopic`. The parameter `expression_parameters` is a handle to a sequence of strings with the parameters used in the SQL expression. The number of parameters in `expression_parameters` must exactly match the number of %n tokens in the subscription expression associated with the `MultiTopic`. This is the subscription expression specified when the `MultiTopic` was created.

Return Code

When the operation returns:

- *RETCODE_OK* - the new expression parameters are set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the number of parameters in `expression_parameters` does not match the number of "%n" tokens in the expression for this `MultiTopic` or one of the parameters is an illegal parameter.
- *RETCODE_ALREADY_DELETED* - the `MultiTopic` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.3.5 TopicListener interface

Since a `Topic` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `TopicListener`. This interface must be implemented by the application. A user-defined class must be provided by the application which must extend to the `TopicListener` interface. **All** `TopicListener` operations **must** be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

⚠ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The `TopicListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as an inconsistent `Topic`. The `TopicListener` is related to changes in communication status.

The interface description is as follows:

```
 public interface TopicListener
 {
 //
 // abstract external operations
 //
    public void
       on_inconsistent_topic
          (Topic the_topic,
           InconsistentTopicStatus status);
 //
 // implemented API operations
 //        <no operations>
 //
 };
```

The next paragraph describes the usage of the `TopicListener` operation. This abstract operation is fully described since it must be implemented by the application.

3.3.5.1 **on_inconsistent_topic (abstract)**

Scope

```
DDS.TopicListener
```

Synopsis

```
import DDS.*;
public void
   on_inconsistent_topic
      (Topic the_topic,
        InconsistentTopicStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `InconsistentTopicStatus` changes.

Parameters

in Topic the_topic - contain a reference to the `Topic` on which the conflict occurred (this is an input to the application).

in   InconsistentTopicStatus   status - contain the `InconsistentTopicStatus` object (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `InconsistentTopicStatus` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `TopicListener` is installed and enabled for the `InconsistentTopicStatus`. The `InconsistentTopicStatus` will change when another `Topic` exists with the same `topic_name` but different characteristics.

The Data Distribution Service will call the `TopicListener` operation with a parameter `the_topic`, which will contain a reference to the `Topic` on which the conflict occurred and a parameter `status`, which will contain the object of the class `InconsistentTopicStatus`.

3.3.6 **ExtTopicListener interface**

The `ExtTopicListener` interface is a subtype of `TopicListener` and provides an OpenSplice-specific callback `on_all_disposed_data`.

> ⚠ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The interface description of this class is as follows:

```
public interface ExtTopicListener extends interface TopicListener
{
// abstract external operations
public void
   on_all_data_disposed
      (Topic the_topic);
//
// implemented API operations
// <no operations>
//
};
```

3.3.6.1 **on_all_data_disposed (abstract)**

Scope

```
DDS.ExtTopicListener
```

Synopsis

```
import DDS.*
   void
      on_all_data_disposed(DDS.Topic the_topic);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the node has completed disposal of data as a result of a call to `Topic.dispose_all_data()`.

Parameters

in DDS.Topic the_topic - contains a reference to the Topic object which has been disposed.

Return Value

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the node has completed disposal of data as a result of a call to `Topic.dispose_all_data()`.

The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `ExtTopicListener` is installed.

Concurrency

The threading behaviour of calls to this method are undefined, so:

- Subsequent disposal via `Topic.dispose_all_data`, and the associated callbacks may be blocked until this method returns.

- This method may be called concurrently by OpenSplice if other `dispose_all_data` operations complete before this method returns.

3.3.7 Topic-Definition type specific interfaces

This paragraph describes the generic `TypeSupport` interfaces and the derived application type specific `<type>TypeSupport` classes which together implement the application `Topic` interface. For each application type, used as `Topic` data type, the pre-processor generates a `<type>DataReader` interface from an IDL type description. The `FooTypeSupport` interface that would be generated by the pre-processor for a fictional type `Foo` describes the `<type>TypeSupport` interfaces.

3.3.7.1 Interface TypeSupport

The `Topic`, `MultiTopic` or `ContentFilteredTopic` is bound to a data type described by the type name argument. Prior to creating a `Topic`, `MultiTopic` or `ContentFilteredTopic`, the data type must have been registered with the Data Distribution Service. This is done using the data type specific `register_type` operation on a extended interface of the `TypeSupport` interface. A extended interface is generated for each data type used by the application, by calling the pre-processor.

The interface description is as follows:

```
public interface TypeSupport
{
//
//  operations
//
// public int
//     register_type
//         (Domainparticipant domain,
//           String type_name);
```

```
//
// public String
//    get_type_name
//         (void);
//
// implemented API operations
//        <no operations>
//
};
```

The next paragraph list the `TypeSupport` operation. This abstract operation is listed but not fully described since it is not implemented in this class. The full description of this operation is given in the `FooTypeSupport` class (for the data type example `Foo`), which contains the data type specific implementation of this operation.

3.3.7.2 get_type_name

This operation is defined as a generic operation, which is implemented by the `<type>TypeSupport` class. For further explanation see the description for the fictional data type `Foo` inherited from the `TypeSupport` class.

Synopsis

```
import DDS.*;
public String
   get_type_name
       (void);
```

3.3.7.3 register_type

This operation is defined as a generic operation, which is implemented by the `<type>TypeSupport` class. For further explanation see the description for the fictional data type `Foo` inherited from the `TypeSupport` class.

Synopsis

```
import DDS.*;
public int
   register_type
       (Domainparticipant domain,
         String type_name);
```

3.3.7.4 Class FooTypeSupport

The pre-processor generates from IDL type descriptions the application `<type>TypeSupport` classes. For each application data type that is used as `Topic` data type, a typed class `<type>TypeSupport` which implements the

TypeSupport interface. In this paragraph, the class FooTypeSupport describes the operations of these <type>TypeSupport interfaces as an example for the fictional application type Foo (defined in the module SPACE).

The Topic, MultiTopic or ContentFilteredTopic is bound to a data type described by the type_name argument. Prior to creating a Topic, MultiTopic or ContentFilteredTopic, the data type must have been registered with the Data Distribution Service. This is done using the data type specific register_type operation on the <type>TypeSupport class for each data type. A class is generated for each data type used by the application, by calling the pre-processor.

The interface description of this class is as follows:

```
public class FooTypeSupport
{
//
// implemented API operations
//
    public int
       register_type
          (DomainParticipant domain,
            String type_name);
    public String
       get_type_name
          (void);
};
```

The next paragraph describes the usage of the FooTypeSupport operation.

3.3.7.5 **get_type_name**

Scope

```
SPACE.FooTypeSupport
```

Synopsis

```
import DDS.*;
public String
   get_type_name
      (void);
```

Description

This operation returns the default name of the data type associated with the FooTypeSupport.

Parameters

Return Value

String - return value is the name of the data type of the `FooTypeSupport`.

Detailed Description

This operation returns the default name of the data type associated with the `FooTypeSupport`. The default name is derived from the type name as specified in the IDL definition. It is composed of the scope names and the type name, each separated by ".", in order of lower scope level to deeper scope level followed by the type name.

3.3.7.6 **register_type**

Scope

```
SPACE.FooTypeSupport
```

Synopsis

```
import DDS.*;
public int
    register_type
        (DomainParticipant domain,
          String type_name);
```

Description

This operation registers a new data type name to a `DomainParticipant`.

Parameters

in Domainparticipant domain - a reference to a `DomainParticipant` object to which the new data type is registered.

in String type_name - a local alias of the new data type to be registered.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation registers a new data type name to a `DomainParticipant`. This operation informs the Data Distribution Service, in order to allow it to manage the new registered data type. This operation also informs the Data Distribution Service about the key definition, which allows the Data Distribution Service to distinguish different instances of the same data type.

Precondition

A `type_name` cannot be registered with two different `<type>TypeSupport` classes (that is, of a different data type) with the same `DomainParticipant`. When the operation is called on the same `DomainParticipant` with the same `type_name` for a different `<type>TypeSupport` class, the operation returns `RETCODE_PRECONDITION_NOT_MET`. However, it is possible to register the same `<type>TypeSupport` classes with the same `DomainParticipant` and the same or different `type_name` multiple times. All registrations return `RETCODE_OK`, but any subsequent registrations with the same `type_name` are ignored.

Return Code

When the operation returns:

- *RETCODE_OK* - the `FooTypeSupport` class is registered with the new data type name to the `DomainParticipant` or the `FooTypeSupport` class was already registered.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - one or both of the parameters is invalid, the `domain` parameter is a `null` reference, or the parameter `type_name` has zero length.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - this `type_name` is already registered with this `DomainParticipant` for a different `<type>TypeSupport` class.

PRISMTECH

3.4 **Publication Module**
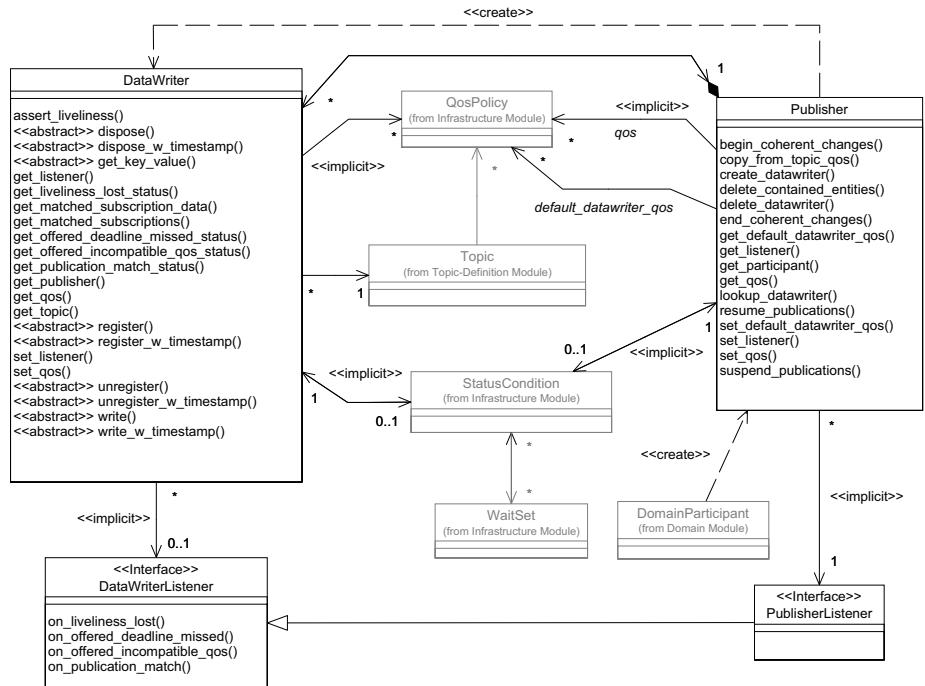
Figure 18 DCPS Publication Module's Class Model

This module contains the following classes:

- Publisher
- Publication type specific classes
- PublisherListener (interface)
- DataWriterListener (interface)

The paragraph "Publication type specific classes" contains the interface and the generated data type specific classes. For each data type, a data type specific class <type>DataWriter is generated (based on IDL) by calling the pre-processor.

For instance, for the fictional data type Foo (this also applies to other types); "Publication type specific classes" contains the following classes:

- DataWriter (abstract)
- FooDataWriter

A `Publisher` is an object responsible for data distribution. It may publish data of different data types. A `DataWriter` acts as a typed accessor to a `Publisher`. The `DataWriter` is the object the application must use to communicate the existence and value of data-objects of a given data type to a `Publisher`. When data-object values have been communicated to the `Publisher` through the appropriate `DataWriter`, it is the `Publisher`'s responsibility to perform the distribution. The `Publisher` will do this according to its own `PublisherQos`, and the `DataWriterQos` attached to the corresponding `DataWriter`. A publication is defined by the association of a `DataWriter` to a `Publisher`. This association expresses the intent of the application to publish the data described by the `DataWriter` in the context provided by the `Publisher`.

3.4.1 Interface Publisher

The `Publisher` acts on behalf of one or more `DataWriter` objects that belong to it. When it is informed of a change to the data associated with one of its `DataWriter` objects, it decides when it is appropriate to actually process the sample-update message. In making this decision, it considers the `PublisherQos` and the `DataWriterQos`.

The interface description of this class is as follows:

```
 public interface Publisher
{
//
// extends interface Entity
//
// public StatusCondition
//    get_statuscondition
//        (void);
// public int
//    get_status_changes
//        (void);
// public int
//    enable
//        (void);
//
// implemented API operations
//
    public DataWriter
       create_datawriter
           (Topic a_topic,
           DataWriterQos qos,
           DataWriterListener a_listener,
           int mask);

    public int
       delete_datawriter
```

PRISMTECH

```
                    (DataWriter a_datawriter);

            public DataWriter
               lookup_datawriter
                  (String topic_name);
            public int
               delete_contained_entities
                  (void);

            public int
               set_qos
                  (PublisherQos qos);
            public int
               get_qos
                  (PublisherQosHolder qos);
            public int
               set_listener
                  (PublisherListener a_listener,
                     int mask);
            public PublisherListener
               get_listener
                  (void);
            public int
               suspend_publications
                  (void);

            public int
               resume_publications
                  (void);

            public int
               begin_coherent_changes
                  (void);

            public int
               end_coherent_changes
                  (void);

            public int
               wait_for_acknowledgments
                  (Duration_t max_wait);

            public DomainParticipant
               get_participant
                  (void);

            public int
               set_default_datawriter_qos
                  (DataWriterQos qos);
```

```
        public int
          get_default_datawriter_qos
              (DataWriterQosHolder qos);

        public int
          copy_from_topic_qos
              (DataWriterQosHolder a_datawriter_qos,
               TopicQos a_topic_qos);
    };
```

The following paragraphs describe the usage of all `Publisher` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the interfaces from which they are inherited.

3.4.1.1 **begin_coherent_changes**

Scope

```
    DDS.Publisher
```

Synopsis

```
    import DDS.*;
    public int
      begin_coherent_changes
          (void);
```

Description

This operation requests that the application will begin a 'coherent set' of modifications using `DataWriter` objects attached to this `Publisher`. The 'coherent set' will be completed by a matching call to `end_coherent_changes`.

Parameters

<none>

Return Value

`int` - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation requests that the application will begin a 'coherent set' of modifications using `DataWriter` objects attached to this `Publisher`. The 'coherent set' will be completed by a matching call to `end_coherent_changes`.

A 'coherent set' is a set of modifications that must be propagated in such a way that they are interpreted at the receivers' 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 precondition for making coherent changes is that the `PresentationQos` of the `Publisher` has its `coherent_access` attribute set to `true`. If this is not the case, the Publisher will not accept any coherent start requests and return `RETCODE_PRECONDITION_NOT_MET`.

A connectivity change may occur in the middle of a set of coherent changes; for example, the set of partitions used by the `Publisher` or one of its connected `Subscribers` may change, a late-joining `DataReader` 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 `end_coherent_changes`.

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).

Return Code

When the operation returns:

- *RETCODE_OK* - a new coherent change has successfully been started.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.
- *RETCODE_PRECONDITION_NOT_MET* - the `Publisher` is not able to handle coherent changes because its `PresentationQos` has not set `coherent_access` to `TRUE`.

3.4.1.2 copy_from_topic_qos

Scope

    DDS.Publisher

Synopsis

    import DDS.*;

```
public int
   copy_from_topic_qos
      (DataWriterQosHolder a_datawriter_qos,
        TopicQos a_topic_qos);
```

Description

This operation will copy policies in a_topic_qos to the corresponding policies in a_datawriter_qos.

Parameters

inout DataWriterQosHolder a_datawriter_qos - the destination DataWriterQos object to which the QosPolicy settings should be copied.

in TopicQos a_topic_qos - the source TopicQos object, which should be copied.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation will copy the QosPolicy settings in a_topic_qos to the corresponding QosPolicy settings in a_datawriter_qos (replacing the values in a_datawriter_qos, if present). This will only apply to the common QosPolicy settings in each <Entity>Qos.

This is a "convenience" operation, useful in combination with the operations get_default_datawriter_qos and Topic.get_qos. The operation copy_from_topic_qos can be used to merge the DataWriter default QosPolicy settings with the corresponding ones on the TopicQos. The resulting DataWriterQos can then be used to create a new DataWriter, or set its DataWriterQos.

This operation does not check the resulting a_datawriter_qos for consistency. This is because the "merged" a_datawriter_qos may not be the final one, as the application can still modify some QosPolicy settings prior to applying the DataWriterQos to the DataWriter.

Return Code

When the operation returns:

- *RETCODE_OK* - the QosPolicy settings are copied from the Topic to the DataWriter.
- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.1.3 create_datawriter

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public DataWriter
   create_datawriter
      (Topic a_topic,
        DataWriterQos qos,
        DataWriterListener a_listener,
        int mask);
```

Description

This operation creates a `DataWriter` with the desired `DataWriterQos`, for the desired `Topic` and attaches the optionally specified `DataWriterListener` to it.

Parameters

in Topic a_topic - a reference to the `topic` for which the `DataWriter` is created.

in DataWriterQos qos - the `DataWriterQos` for the new `DataWriter`. In case these settings are not self consistent, no `DataWriter` is created.

in DataWriterListener a_listener - a reference to the `DataWriterListener` instance which will be attached to the new `DataWriter` It is permitted to use `null` as the value of the listener: this behaves as a `DataWriterListener` whose operations perform no action.

in int mask - a bit-mask in which each bit enables the invocation of the `DataWriterListener` for a certain status.

Return Value

DataWriter - Return value is a reference to the newly created `DataWriter`. In case of an error, the `null` reference is returned.

Detailed Description

This operation creates a `DataWriter` with the desired `DataWriterQos`, for the desired `Topic` and attaches the optionally specified `DataWriterListener` to it. The returned `DataWriter` is attached (and belongs) to the `Publisher` on which

this operation is being called. To delete the `DataWriter` the operation `delete_datawriter` or `delete_contained_entities` must be used. If no write rights are defined for the specific topic then the creation of the `DataWriter` will fail.

Application Data Type

The `DataWriter` returned by this operation is an object of a derived class, specific to the data type associated with the `Topic`. For each application-defined data type `<type>` there is a class `<type>DataWriter` generated by calling the pre-processor. This data type specific class extends `DataWriter` and contains the operations to write data of data type `<type>`.

QosPolicy

The possible application pattern to construct the `DataWriterQos` for the `DataWriter` is to:

• Retrieve the `QosPolicy` settings on the associated `Topic` by means of the `get_qos` operation on the `Topic`

• Retrieve the default `DataWriterQos` by means of the `get_default_datawriter_qos` operation on the `Publisher`

• Combine those two lists of `QosPolicy` settings and selectively modify `QosPolicy` settings as desired

• Use the resulting `DataWriterQos` to construct the `DataWriter`

In case the specified `QosPolicy` settings are not consistent, no `DataWriter` is created and the `null` reference is returned.

Default QoS

The constant `DATAWRITER_QOS_DEFAULT` can be used as parameter `qos` to create a `DataWriter` with the default `DataWriterQos` as set in the `Publisher`. The effect of using `DATAWRITER_QOS_DEFAULT` is the same as calling the operation `get_default_datawriter_qos` and using the resulting `DataWriterQos` to create the `DataWriter`.

The special `DATAWRITER_QOS_USE_TOPIC_QOS` can be used to create a `DataWriter` with a combination of the default `DataWriterQos` and the `TopicQos`. The effect of using `DATAWRITER_QOS_USE_TOPIC_QOS` is the same as calling the operation `get_default_datawriter_qos` and retrieving the `TopicQos` (by means of the operation `Topic.get_qos`) and then combining these two `QosPolicy` settings using the operation `copy_from_topic_qos`, whereby any common policy that is set on the `TopicQos` "overrides" the corresponding policy on the default `DataWriterQos`. The resulting `DataWriterQos` is then applied to create the `DataWriter`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `DataWriterListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `DataWriterListener`:

- `OFFERED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `LIVELINESS_LOST_STATUS`
- `PUBLICATION_MATCHED_STATUS`.

Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

In case a communication status is not activated in the `mask` of the `DataWriterListener`, the `PublisherListener` of the containing `Publisher` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `PublisherListener` of the containing `Publisher` and a `DataWriter` specific behaviour when needed. In case the communication status is not activated in the `mask` of the `PublisherListener` as well, the communication status will be propagated to the `DomainParticipantListener` of the containing `DomainParticipant`. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

3.4.1.4 **delete_contained_entities**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   delete_contained_entities
       (void);
```

Description

This operation deletes all the `DataWriter` objects that were created by means of one of the `create_datawriter` operations on the `Publisher`.

Parameters

<none>

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.
```

Detailed Description

This operation deletes all the `DataWriter` objects that were created by means of one of the `create_datawriter` operations on the `Publisher`. In other words, it deletes all contained `DataWriter` objects.

⚠️ **NOTE***: The operation will return `PRECONDITION_NOT_MET` if the any of the contained entities is in a state where it cannot be deleted. In such cases, the operation does not roll back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

• *RETCODE_OK* - the contained `Entity` objects are deleted and the application may delete the `Publisher`

• *RETCODE_ERROR* - an internal error has occurred.

• *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.

• *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - one or more of the contained entities are in a state where they cannot be deleted.

3.4.1.5 delete_datawriter

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   delete_datawriter
      (DataWriter a_datawriter);
```

Description

This operation deletes a `DataWriter` that belongs to the `Publisher`.

Parameters

in DataWriter a_datawriter - a reference to the `DataWriter`, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes a `DataWriter` that belongs to the `Publisher`. When the operation is called on a different `Publisher`, as used when the `DataWriter` was created, the operation has no effect and returns `RETCODE_PRECONDITION_NOT_MET`. The deletion of the `DataWriter` will automatically unregister all instances. Depending on the settings of `WriterDataLifecycleQosPolicy`, the deletion of the `DataWriter` may also dispose of all instances.

Return Code

When the operation returns:

- *RETCODE_OK* - the `DataWriter` is deleted

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter a_datawriter is not a valid `DataWriter` reference.

- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `Publisher`, as used when the `DataWriter` was created.

3.4.1.6 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   enable
      (void);
```

3.4.1.7 end_coherent_changes

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   end_coherent_changes
      (void);
```

Description

This operation terminates the 'coherent set' initiated by the matching call to `begin_coherent_changes`.

Parameters

<none>

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation terminates the 'coherent set' initiated by the matching call to
`begin_coherent_changes`. If there is no matching call to
`begin_coherent_changes`, the operation will return the error
`PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the coherent change has successfully been closed.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.
- *RETCODE_PRECONDITION_NOT_MET* - there is no matching `begin_coherent_` `changes` call that can be closed.

3.4.1.8 **get_default_datawriter_qos**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   get_default_datawriter_qos
       (DataWriterQosHolder qos);
```

Description

This operation gets the default `DataWriterQos` of the `Publisher`.

Parameters

inout DataWriterQosHolder qos - a reference to the destination
   `DataWriterQosHolder` object in which the default `DataWriterQos` for the
   `Publisher` is written.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or
   `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets the default `DataWriterQos` of the `Publisher` (that is the object with the `QosPolicy` settings) which is used for newly created `DataWriter` objects, in case the constant `DATAWRITER_QOS_DEFAULT` is used. The default `DataWriterQos` is only used when the constant is supplied as parameter `qos` to specify the `DataWriterQos` in the `create_datawriter` operation. The application must provide the `DataWriterQos` object in which the `QosPolicy` settings can be stored and pass the `qos` reference to the operation. The operation writes the default `DataWriterQos` to the object referenced to by `qos`. Any settings in the object are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_datawriter_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` setting.

Return Code

When the operation returns:

- *RETCODE_OK* - the default `DataWriter` `QosPolicy` settings of this `Publisher` have successfully been copied into the specified `DataWriterQosHolder` parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.1.9 get_listener

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public PublisherListener
   get_listener
      (void);
```

Description

This operation allows access to a `PublisherListener`.

Parameters

Return Value

PublisherListener - result is a reference to the `PublisherListener` attached to the `Publisher`.

Detailed Description

This operation allows access to a `PublisherListener` attached to the `Publisher`. When no `PublisherListener` was attached to the `Publisher`, the `null` reference is returned.

3.4.1.10 **get_participant**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public DomainParticipant
   get_participant
      (void);
```

Description

This operation returns the `DomainParticipant` associated with the `Publisher` or the `null` reference.

Parameters

<none>

Return Value

DomainParticipant - a reference to the `DomainParticipant` associated with the `Publisher` or the `null` reference.

Detailed Description

This operation returns the `DomainParticipant` associated with the `Publisher`. Note that there is exactly one `DomainParticipant` associated with each `Publisher`. When the `Publisher` was already deleted (there is no associated `DomainParticipant` any more), the `null` reference is returned.

3.4.1.11 **get_qos**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
    get_qos
        (PublisherQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a `Publisher`.

Parameters

inout PublisherQosHolder qos - the destination `PublisherQosHolder`
    object in which the `QosPolicy` settings will be copied.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or
    `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `Publisher` on
which this operation is used. This `PublisherQos` is stored in the `qos` parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this `Publisher`
  has successfully been copied into the specified `PublisherQosHolder`
  parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.4.1.12 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class `Entity`
for further explanation.

Synopsis

```
import DDS.*;
public int
    get_status_changes
        (void);
```

3.4.1.13 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
       (void);
```

3.4.1.14 **lookup_datawriter**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public DataWriter
   lookup_datawriter
       (String topic_name);
```

Description

This operation returns a previously created `DataWriter` belonging to the `Publisher` which is attached to a `Topic` with the matching `topic_name`.

Parameters

in String topic_name - the name of the `Topic`, which is attached to the `DataWriter` to look for.

Return Value

DataWriter - Return value is a reference to the `DataWriter` found. When no such `DataWriter` is found, the `null` reference is returned.

Detailed Description

This operation returns a previously created `DataWriter` belonging to the `Publisher` which is attached to a `Topic` with the matching `topic_name`. When multiple `DataWriter` objects (which satisfy the same condition) exist, this operation will return one of them. It is not specified which one.

3.4.1.15 **resume_publications**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
    resume_publications
        (void);
```

Description

This operation resumes a previously suspended publication.

Parameters

<none>

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED or
RETCODE_PRECONDITION_NOT_MET.

Detailed Description

If the `Publisher` is suspended, this operation will resume the publication of all `DataWriter` objects contained by this `Publisher`. All data held in the history buffer of the `DataWriter`'s is actively published to the consumers. When the operation returns all `DataWriter`'s have resumed the publication of suspended updates.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Publisher` has been suspended.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `Publisher` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - the `Publisher` is not suspended.

3.4.1.16 set_default_datawriter_qos

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   set_default_datawriter_qos
      (DataWriterQos qos);
```

Description

This operation sets the default `DataWriterQos` of the `Publisher`.

Parameters

in DataWriterQos qos - an instance of the `DataWriterQos` class, which contains the new default `DataWriterQos` for the newly created `DataWriters`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`,   `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_INCONSISTENT_POLICY`.

Detailed Description

This operation sets the default `DataWriterQos` of the `Publisher` (that is the compound class with the `QosPolicy` settings) which is used for newly created `DataWriter` objects, in case the constant `DATAWRITER_QOS_DEFAULT` is used. The default `DataWriterQos` is only used when the constant is supplied as parameter qos to specify the `DataWriterQos` in the `create_datawriter` operation. The `set_default_datawriter_qos` operation checks if the `DataWriterQos` is self consistent. If it is not, the operation has no effect and returns `RETCODE_INCONSISTENT_POLICY`.

The values set by this operation are returned by `get_default_datawriter_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `DataWriterQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid `DataWriterQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value.

- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted

- *RETCODE_INCONSISTENT_POLICY* - the parameter `qos` contains conflicting `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.4.1.17 set_listener

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   set_listener
      (PublisherListener a_listener,
        int mask);
```

Description

This operation attaches a `PublisherListener` to the `Publisher`.

Parameters

in PublisherListener a_listener - a reference to the `PublisherListener` instance, which will be attached to the `Publisher`.

in int mask - a bit mask in which each bit enables the invocation of the `PublisherListener` for a certain status.

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK,   RETCODE_ERROR,   RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED or RETCODE_OUT_ OF_RESOURCES.

Detailed Description

This operation attaches a `PublisherListener` to the `Publisher`. Only one `PublisherListener` can be attached to each `Publisher`. If a `PublisherListener` was already attached, the operation will replace it with the new one. When a_listener is the `null` reference, it represents a listener that is treated as a NOOP[1] for all statuses activated in the bitmask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `PublisherListener`

---

1.   Short for **No-Op**eration, an instruction that peforms nothing at all.

operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset. An exception to this rule is the `null` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `PublisherListener`:

- `OFFERED_DEADLINE_MISSED_STATUS`   *(propagated)*
- `OFFERED_INCOMPATIBLE_QOS_STATUS`   *(propagated)*
- `LIVELINESS_LOST_STATUS`   *(propagated)*
- `PUBLICATION_MATCHED_STATUS`   *(propagated)*.

Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataWriterListener` of a contained `DataWriter`, the `DataWriterListener` on that contained `DataWriter` is invoked instead of the `PublisherListener`. This means that a status change on a contained `DataWriter` only invokes the `PublisherListener` if the contained `DataWriter` itself does not handle the trigger event generated by the status change.

In case a status is not activated in the `mask` of the `PublisherListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Publisher` specific behaviour when

needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

Return Code

- *RETCODE_OK* - the `PublisherListener` is attached.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_UNSUPPORTED* - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.

- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.1.18 set_qos

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
    set_qos
        (PublisherQos qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `Publisher`.

Parameters

in PublisherQos qos - the new set of `QosPolicy` settings for the `Publisher`.

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR,  RETCODE_BAD_PARAMETER,
RETCODE_UNSUPPORTED,     RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES, RETCODE_IMMUTABLE_POLICY or
RETCODE_PRECONDITION_NOT_MET.
```

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `Publisher`. The parameter `qos` contains the `QosPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QosPolicy`

setting for an enabled `Publisher`, which can only be set before the `Publisher` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the currently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` settings (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

The set of `QosPolicy` settings specified by the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned `RETCODE_OK`). If one or more of the partitions in the QoS structure have insufficient access rights configured then the `set_qos` function will fail with a `RETCODE_PRECONDITION_NOT_MET` error code.

Return Code

When the operation returns:

- *RETCODE_OK* - the new `PublisherQos` is set.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `qos` is not a valid `PublisherQos`.
- `RETCODE_UNSUPPORTED` - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter `qos` contains an immutable `QosPolicy` setting with a different value than set during enabling of the `Publisher`.
- *RETCODE_PRECONDITION_NOT_MET* - returned when insufficient access rights exist for the partition(s) listed in the `QoS` structure.

3.4.1.19 **suspend_publications**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   suspend_publications
       (void);
```

Description

This operation will suspend the dissemination of the publications by all contained `DataWriter` objects.

Parameters

<none>

Return Value

`int` - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation suspends the publication of all `DataWriter` objects contained by this `Publisher`. The data written or disposed by a `DataWriter` is stored in the history buffer of the `DataWriter` and therefore, depending on its QoS settings, the following operations may block (see the operation descriptions for more information):

- `DataWriter.dispose`
- `DataWriter.dispose_w_timestamp`
- `DataWriter.write`
- `DataWriter.write_w_timestamp`
- `DataWriter.writedispose`
- `DataWriter.writedispose_w_timestamp`
- `DataWriter.unregister_instance`
- `DataWriter.unregister_instance_w_timestamp`

Subsequent calls to this operation have no effect. When the `Publisher` is deleted before `resume_publication` is called, all suspended updates are discarded.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Publisher` has been suspended
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `Publisher` is not enabled.

PRISMTECH

3.4.1.20 **wait_for_acknowledgments**

Scope

```
DDS.Publisher
```

Synopsis

```
import DDS.*;
public int
   wait_for_acknowledgments
      (Duration_t max_wait);
```

Description

This operation blocks the calling thread until either all data written by all contained `DataWriters` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Parameters

in Duration_t max_wait - the maximum duration to block for the `wait_for_acknowledgments`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

int - Possible return codes of the operation are:
  `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,`
  `RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED` or
  `RETCODE_TIMEOUT.`

Detailed Description

This operation blocks the calling thread until either all data written by all contained DataWriters is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Data is acknowledged by the local infrastructure when it does not need to be stored in its DataWriter's local history. When a locally-connected subscription (including the networking service) has no more resources to store incoming samples it will start to reject these samples, resulting in their source DataWriters to store them temporarily in their own local history to be retransmitted at a later moment in time. In such scenarios, the `wait_for_acknowledgments` operation will block until all contained DataWriters have retransmitted their entire history, which is therefore effectively empty, or until the `max_wait` timeout expires, whichever happens first. In the first case the operation will return `RETCODE_OK`, in the latter it will return `RETCODE_TIMEOUT`.

⚠ Be aware that in case the operation returns RETCODE_OK, the data has only been acknowledged by the local infrastructure: it does not mean all remote subscriptions have already received the data. However, delivering the data to remote nodes is then the sole responsibility of the networking service: even when the publishing application would terminate, all data that has not yet been received may be considered 'on-route' and will therefore eventually arrive (unless the networking service itself will crash). In contrast, if a DataWriter would still have data in its local history buffer when it terminates, this data is considered 'lost'.

This operation is intended to be used only if one or more of the contained DataWriters has its ReliabilityQosPolicyKind set to RELIABLE_RELIABILITY_QOS. Otherwise the operation will return immediately with RETCODE_OK, since best-effort DataWriters will never store rejected samples in their local history: they will just drop them and continue business as usual.

Return Code

When the operation returns:

- *RETCODE_OK* - the data of all contained DataWriters has been acknowledged by the local infrastructure.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the Publisher has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the Publisher is not enabled.
- *RETCODE_TIMEOUT* - not all data is acknowledged before max_wait elapsed.

3.4.2 Publication Type Specific Classes

This paragraph describes the generic DataWriter class and the derived application type specific <type>DataWriter classes which together implement the application publication interface. For each application type, used as Topic data type, the pre-processor generates a <type>DataWriter class from an IDL type description. The FooDataWriter class that would be generated by the pre-processor for a fictional type Foo describes the <type>DataWriter classes.

3.4.2.1 Interface DataWriter

DataWriter allows the application to set the value of the sample to be published under a given Topic.

A DataWriter is attached to exactly one Publisher which acts as a factory for it.

A DataWriter is bound to exactly one Topic and therefore to exactly one data type. The Topic must exist prior to the DataWriter's creation.

DataWriter is an interface. It must be extended for each particular application data type. For a fictional application data type Foo (defined in the module SPACE) the extended interface would be SPACE.FooDataWriter.

The interface description is as follows:

```
 public interface DataWriter
{
//
// extends interface Entity
//
// public StatusCondition
//     get_statuscondition
//         (void);
// public int
//     get_status_changes
//         (void);
// public int
//     enable
//         (void);
//
// abstract operations (implemented in the data type specific
//  DataWriter)
//
// public long
//     register_instance
//         (<data> instance_data);
//
// public long
//     register_instance_w_timestamp
//         (<data> instance_data,
//           Time_t source_timestamp);
//
// public int
//     unregister_instance
//         (<data> instance_data,
//           long handle);
//
// public int
//     unregister_instance_w_timestamp
//         (<data> instance_data,
//           long handle,
//           Time_t source_timestamp);
//
// public int
//     write
//         (<data> instance_data,
//           long handle);
//
// public int
```

```
//      write_w_timestamp
//          (<data> instance_data,
//            long handle,
//            Time_t source_timestamp);
//
// public int
//     dispose
//          (<data> instance_data,
//            long instance_handle);
//
// public int
//     dispose_w_timestamp
//          (<data> instance_data,
//            long instance_handle,
//            Time_t source_timestamp);
//
// public int
//     writedispose
//          (<data> instance_data,
//            long instance_handle);
//
// public int
//     writedispose_w_timestamp
//          (<data> instance_data,
//            long instance_handle,
//            Time_t source_timestamp);
//
// public int
//     get_key_value
//          (<data>Holder key_holder,
//            long handle);
//
// public long
//     lookup_instance
//          (<data> instance_data);
//
// implemented API operations
//
    public int
       set_qos
           (DataWriterQos qos);

    public int
       get_qos
           (DataWriterQosHolder qos);

    public int
       set_listener
           (DataWriterListener a_listener,
            int mask);
```

PRISMTECH

```
                    public DataWriterListener
                       get_listener
                          (void);

                    public Topic
                       get_topic
                          (void);

                    public Publisher
                       get_publisher
                          (void);

                    public int
                       wait_for_acknowledgments
                          (Duration_t max_wait);

                    public int
                       get_liveliness_lost_status
                          (LivelinessLostStatusHolder status);

                    public int
                       get_offered_deadline_missed_status
                          (OfferedDeadlineMissedStatusHolder status);

                    public int
                       get_offered_incompatible_qos_status
                          (PublicationMatchedStatusHolder status);

                    public int
                       get_publication_matched_status
                          (PublicationMatchedStatusHolder status);

                    public int
                       assert_liveliness
                          (void);

                    public int
                       get_matched_subscriptions
                          (InstanceHandleSeqHolder subscription_handles);

                    public int
                       get_matched_subscription_data
                          (SubscriptionBuiltinTopicDataHolder subscription_data,
                          long subscription_handle);
                 };
```

The following paragraphs describe the usage of all `DataWriter` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the

classes from which they are inherited. The abstract operations are listed but not fully described because they are not implemented in this specific class. The full description of these operations is located in the subclasses, which contain the data type specific implementation of these operations.

3.4.2.2 **assert_liveliness**

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
   assert_liveliness
       (void);
```

Description

This operation asserts the liveliness for the `DataWriter`.

Parameters

<none>

Return Value

int - Possible return codes of the operation are:
  `RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,`
  `RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation will manually assert the liveliness for the `DataWriter`. This way, the Data Distribution Service is informed that the corresponding `DataWriter` is still alive. This operation is used in combination with the `LivelinessQosPolicy` set to `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS` or `MANUAL_BY_TOPIC_LIVELINESS_QOS`, See Section 3.1.3.10, *LivelinessQosPolicy*, on page 62 for more information on `LivelinessQosPolicy`.

Writing data via the `write` operation of a `DataWriter` will assert the liveliness on the `DataWriter` itself and its containing `DomainParticipant`. Therefore, `assert_liveliness` is only needed when **not** writing regularly.

The liveliness should be asserted by the application, depending on the `LivelinessQosPolicy`. Asserting the liveliness for this `DataWriter` can also be achieved by asserting the liveliness to the `DomainParticipant`.

PRISMTECH

Return Code

When the operation returns:

- *RETCODE_OK* - the liveliness of this `DataWriter` has successfully been asserted.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `DataWriter` is not enabled.

3.4.2.3 **dispose (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
   dispose
      (<data> instance_data,
        long instance_handle);
```

3.4.2.4 **dispose_w_timestamp (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
   dispose_w_timestamp
      (<data> instance_data,
        long instance_handle,
        Time_t source_timestamp);
```

3.4.2.5 **enable (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
    enable
        (void);
```

3.4.2.6 get_key_value (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
    get_key_value
       (<data>Holder key_holder,
         long handle);
```

3.4.2.7 get_listener

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public DataWriterListener
    get_listener
        (void);
```

Description

This operation allows access to a `DataWriterListener`.

Parameters

<none>

Return Value

DataWriterListener - result is a reference to the `DataWriterListener` attached to the `DataWriter`.

Detailed Description

This operation allows access to a `DataWriterListener` attached to the `DataWriter`. When no `DataWriterListener` was attached to the `DataWriter`, the `null` reference is returned.

3.4.2.8 get_liveliness_lost_status

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
    get_liveliness_lost_status
        (LivelinessLostStatusHolder status);
```

Description

This operation obtains a `LivelinessLostStatus` object of the `DataWriter`.

Parameters

inout LivelinessLostStatusHolder status - the contents of the `LivelinessLostStatus` object of the `DataWriter` will be copied into the `LivelinessLostStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `LivelinessLostStatus` object of the `DataWriter`. This object contains the information whether the liveliness (that the `DataWriter` has committed through its `LivelinessQosPolicy`) was respected.

This means that the status represents whether the `DataWriter` failed to actively signal its liveliness within the offered liveliness period. If the liveliness is lost, the `DataReader` objects will consider the `DataWriter` as no longer "alive".

The `LivelinessLostStatus` can also be monitored using a `DataWriterListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current LivelinessLostStatus of this DataWriter has successfully been copied into the specified status parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the DataWriter has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.9 get_matched_subscription_data

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
   get_matched_subscription_data
      (SubscriptionBuiltinTopicDataHolder subscription_data,
       long subscription_handle);
```

Description

This operation retrieves information on the specified subscription that is currently "associated" with the DataWriter.

Parameters

inout SubscriptionBuiltinTopicDataHolder subscription_data - a Holder for the sample in which the information about the specified subscription is to be stored.

in long subscription_handle - a handle to the subscription whose information needs to be retrieved.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED, RETCODE_OUT_ OF_RESOURCES or RETCODE_NOT_ENABLED.

Detailed Description

This operation retrieves information on the specified subscription that is currently "associated" with the `DataWriter`. That is, a subscription with a matching Topic and compatible QoS that the application has not indicated should be "ignored" by means of the `ignore_subscription` operation on the `DomainParticipant` class.

The `subscription_handle` must correspond to a subscription currently associated with the `DataWriter`, otherwise the operation will fail and return `RETCODE_BAD_PARAMETER`. The operation `get_matched_subscriptions` can be used to find the subscriptions that are currently matched with the `DataWriter`.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the `subscription_data`. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In such cases the operation will return `RETCODE_UNSUPPORTED`.

Return Code

When the operation returns:

- *RETCODE_OK* - the information on the specified subscription has successfully been retrieved.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" subscriptions.

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `DataWriter` is not enabled.

3.4.2.10 get_matched_subscriptions

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
  get_matched_subscriptions
      (InstanceHandleSeqHolder subscription_handles);
```

Description

This operation retrieves the list of subscriptions currently "associated" with the `DataWriter`.

Parameters

inout InstanceHandleSeqHolder subscription_handles - a Holder for an array which is used to pass the list of all associated subscribtions.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED, RETCODE_OUT_ OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation retrieves the list of subscriptions currently "associated" with the `DataWriter`. That is, subscriptions that have a matching Topic and compatible QoS that the application has not indicated should be "ignored" by means of the `ignore_subscription` operation on the `DomainParticipant` class.

The array inside the `subscription_handles` Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the `get_matched_subscriptions` operation. If the pre-allocated array is not big enough to hold the number of associated subscriptions, the array will automatically be (re-)allocated to fit the required size.

The handles returned in the `subscription_handles` array are the ones that are used by the DDS implementation to locally identify the corresponding matched subscription entities. You can access more detailed information about a particular subscription by passing its `subscription_handle` to either the `get_matched_subscription_data` operation or to the `read_instance` operation on the built-in reader for the "DCPSSubscription" topic.

Be aware that since an instance handle is an opaque datatype, it does not necessarily mean that the handles obtained from the `get_matched_subscriptions` operation have the same value as the ones that appear in the `instance_handle` field of the `SampleInfo` when retrieving the subscription info through corresponding "DCPSSubscriptions" built-in reader. You can't just compare two handles to determine whether they represent the same subscription. If you want to know whether two handles actually do represent the same subscription, use both handles to retrieve their corresponding `SubscriptionBuiltinTopicData` samples and then compare the `key` field of both samples.

PRISMTECH

The operation may fail if the infrastructure does not locally maintain the connectivity information. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In such cases the operation will return `RETCODE_UNSUPPORTED`.

Return Code

When the operation returns:

- *RETCODE_OK* - the list of associated subscriptions has successfully been obtained.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" subscriptions.
- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `DataWriter` is not enabled.

3.4.2.11 **get_offered_deadline_missed_status**

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
   get_offered_deadline_missed_status
      (OfferedDeadlineMissedStatusHolder status);
```

Description

This operation obtains the `OfferedDeadlineMissedStatus` object of the `DataWriter`.

Parameters

inout OfferedDeadlineMissedStatusHolder status - the contents of the `OfferedDeadlineMissedStatus` object of the `DataWriter` will be copied into the `OfferedDeadlineMissedStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or
RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation obtains the `OfferedDeadlineMissedStatus` object of the
`DataWriter`. This object contains the information whether the deadline (that the
`DataWriter` has committed through its `DeadlineQosPolicy`) was respected for
each instance.

The `OfferedDeadlineMissedStatus` can also be monitored using a
`DataWriterListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `LivelinessLostStatus` of this `DataWriter` has
  successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.4.2.12 get_offered_incompatible_qos_status

Scope

DDS.DataWriter

Synopsis

```
import DDS.*;
public int
    get_offered_incompatible_qos_status
        (OfferedIncompatibleQosStatusHolder status);
```

Description

This operation obtains the `OfferedIncompatibleQosStatus` object.

Parameters

inout OfferedIncompatibleQosStatusHolder status - the contents of
the `OfferedIncompatibleQosStatus` object of the `DataWriter` will be
copied into the `OfferedIncompatibleQosStatusHolder` specified by
`status`.

Return Value

int - Possible return codes of the operation are:
  RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or
  RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation obtains the OfferedIncompatibleQosStatus object of the
DataWriter. This object contains the information whether a QosPolicy setting
was incompatible with the requested QosPolicy setting.

This means that the status represents whether a DataReader object has been
discovered by the DataWriter with the same Topic and a requested
DataReaderQos that was incompatible with the one offered by the DataWriter.

The OfferedIncompatibleQosStatus can also be monitored using a
DataWriterListener or by using the associated StatusCondition.

Return Code

When the operation returns:

• *RETCODE_OK* - the current OfferedIncompatibleQosStatus of this
  DataWriter has successfully been copied into the specified status parameter.

• *RETCODE_ERROR* - an internal error has occurred.

• *RETCODE_ALREADY_DELETED* - the DataWriter has already been deleted.

• *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.4.2.13 get_publication_matched_status

Scope

    DDS.DataWriter

Synopsis

    import DDS.*;
    public int
      get_publication_match_status
        (PublicationMatchedStatusHolder status);

Description

This operation obtains the PublicationMatchedStatus object of the
DataWriter.

Parameters

inout PublicationMatchedStatusHolder status - the contents of the
   `PublicationMatchedStatus` object of the `DataWriter` will be copied into
   the `PublicationMatchedStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`,
   `RETCODE_ALREADY_DELETED` or `RETCODE_ OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `PublicationMatchedStatus` object of the
`DataWriter`. This object contains the information whether a new match has been
discovered for the current publication, or whether an existing match has ceased to
exist.

This means that the status represents that either a DataReader object has been
discovered by the `DataWriter` with the same Topic and a compatible Qos, or that a
previously discovered DataReader has ceased to be matched to the current
`DataWriter`. A DataReader may cease to match when it gets deleted, when it
changes its Qos to a value that is incompatible with the current `DataWriter` or
when either the `DataWriter` or the DataReader has chosen to put its matching
counterpart on its ignore-list using the `ignore_subcription` or
`ignore_publication` operations on the `DomainParticipant`.

The operation may fail if the infrastructure does not hold the information necessary
to fill in the `PublicationMatchedStatus`. This is the case when OpenSplice is
configured not to maintain discovery information in the Networking Service. (See
the description for the `NetworkingService/Discovery/enabled` property in
the Deployment Manual for more information about this subject.) In this case the
operation will return `RETCODE_UNSUPPORTED`.

The `PublicationMatchedStatus` can also be monitored using a
`DataWriterListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `PublicationMatchedStatus` of this `DataWriter`
  has successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the
  information about "associated" subscriptions.

PRISMTECH

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.14 get_publisher

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public Publisher
   get_publisher
       (void);
```

Description

This operation returns the `Publisher` to which the `DataWriter` belongs.

Parameters

<none>

Return Value

Publisher - Return value is a reference to the `Publisher` object to which the `DataWriter` belongs.

Detailed Description

This operation returns the `Publisher` to which the `DataWriter` belongs, thus the `Publisher` that has created the `DataWriter`. If the `DataWriter` is already deleted, the `null` reference is returned.

3.4.2.15 get_qos

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
   get_qos
       (DataWriterQosHolder qos);
```

Description

This operation allows access to the existing list of `QosPolicy` settings for a `DataWriter`.

Parameters

inout DataWriterQosHolder qos - the destination `PublisherQosHolder` object in which the `QosPolicy` settings will be copied.

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing list of `QosPolicy` settings of a `DataWriter` on which this operation is used. This `DataWriterQos` is stored at the location referenced by the `qos` parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of `QosPolicy` values applied to this `DataWriter` has successfully been copied into the specified `DataWriterQosHolder` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.16 **get_status_changes (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
       (void);
```

3.4.2.17 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

PRISMTECH

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
       (void);
```

3.4.2.18 get_topic

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public Topic
   get_topic
       (void);
```

Description

This operation returns the `Topic` object which is associated with the `DataWriter`.

Parameters

<none>

Return Value

Topic - Return value is a reference to the `Topic` which is associated with the `DataWriter`.

Detailed Description

This operation returns the `Topic` which is associated with the `DataWriter`, thus the `Topic` with which the `DataWriter` is created. If the `DataWriter` is already deleted, the `null` reference is returned.

3.4.2.19 lookup_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public long
   lookup_instance
```

```
                    (<data> instance_data);
```

3.4.2.20 register_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by
the `<type>DataWriter` class. Therefore, to use this operation, the data type
specific implementation of this operation in its respective derived class must be
used. For further explanation see the description for the fictional data type `Foo`
derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public long
   register_instance
      (<data> instance_data);
```

3.4.2.21 register_instance_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by
the `<type>DataWriter` class. Therefore, to use this operation, the data type
specific implementation of this operation in its respective derived class must be
used. For further explanation see the description for the fictional data type `Foo`
derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public long
   register_instance_w_timestamp
      (<data> instance_data,
        Time_t source_timestamp);
```

3.4.2.22 set_listener

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
   set_listener
      (DataWriterListener a_listener,
        int mask);
```

Description

This operation attaches a `DataWriterListener` to the `DataWriter`.

Parameters

in DataWriterListener a_listener - a reference to the DataWriterListener instance, which will be attached to the DataWriter.

in int mask - a bit mask in which each bit enables the invocation of the DataWriterListener for a certain status.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED or RETCODE_OUT_ OF_RESOURCES.

Detailed Description

This operation attaches a DataWriterListener to the DataWriter. Only one DataWriterListener can be attached to each DataWriter. If a DataWriterListener was already attached, the operation will replace it with the new one. When a_listener is the null reference, it represents a listener that is treated as a NOOP[1] for all statuses activated in the bitmask.

Communication Status

For each communication status, the StatusChangedFlag flag is initially set to false. It becomes true whenever that communication status changes. For each communication status activated in the mask, the associated DataWriterListener operation is invoked and the communication status is reset to false, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the get_<status_name>_status from inside the listener it will see the status already reset. An exception to this rule is the null listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the DataWriterListener:

- OFFERED_DEADLINE_MISSED_STATUS
- OFFERED_INCOMPATIBLE_QOS_STATUS
- LIVELINESS_LOST_STATUS
- PUBLICATION_MATCHED_STATUS.

Be aware that the PUBLICATION_MATCHED_STATUS is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the

---

1.   Short for **No-Op**eration, an instruction that peforms nothing at all.

`NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

In case a communication status is not activated in the `mask` of the `DataWriterListener`, the `PublisherListener` of the containing `Publisher` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `PublisherListener` of the containing `Publisher` and a `DataWriter` specific behaviour when needed. In case the communication status is not activated in the `mask` of the `PublisherListener` as well, the communication status will be propagated to the `DomainParticipantListener` of the containing `DomainParticipant`. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

Return Code

When the operation returns:

- *RETCODE_OK* - the `DataWriterListener` is attached
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_UNSUPPORTED* - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.23 set_qos

Scope

```
DDS.DataWriter
```

Synopsis

```
import DDS.*;
public int
```

```
      set_qos
        (DataWriterQos qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `DataWriter`.

Parameters

in DataWriterQos qos - the new set of `QosPolicy` settings for the `DataWriter`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`,   `RETCODE_ALLREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_IMMUTABLE_POLICY` or `RETCODE_INCONSISTENT_POLICY`.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `DataWriter`. The parameter `qos` contains the object with the `QosPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QosPolicy` setting for an enabled `DataWriter`, which can only be set before the `DataWriter` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the presently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` setting (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

The set of `QosPolicy` settings specified by the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned `RETCODE_OK`).

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `DataWriterQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter `qos` is not a valid `DataWriterQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value.

- *RETCODE_UNSUPPORTED* - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted
- *RETCODE_IMMUTABLE_POLICY* - the parameter `qos` contains an immutable `QosPolicy` setting with a different value than set during enabling of the `DataWriter`.
- *RETCODE_INCONSISTENT_POLICY* - the parameter `qos` contains an inconsistent `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.4.2.24 **unregister_instance (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
   unregister_instance
      (<data> instance_data,
        long handle);
```

3.4.2.25 **unregister_instance_w_timestamp (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
   unregister_instance_w_timestamp
      (<data> instance_data,
        long handle,
        Time_t source_timestamp);
```

3.4.2.26 **wait_for_acknowledgments**

Synopsis

```
import DDS.*;
public int
   wait_for_acknowledgments
      (Duration_t max_wait);
```

Description

This operation blocks the calling thread until either all data written by the `DataWriter` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Parameters

`in Duration_t max_wait` - the maximum duration to block for the `wait_for_acknowledgments`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,`
`RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED or`
`RETCODE_TIMEOUT.`

Detailed Description

This operation blocks the calling thread until either all data written by the `DataWriter` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Data is acknowledged by the local infrastructure when it does not need to be stored in its DataWriter's local history. When a locally-connected subscription (including the networking service) has no more resources to store incoming samples it will start to reject these samples, resulting in its source DataWriter to store them temporarily in its own local history to be retransmitted at a later moment in time. In such scenarios, the `wait_for_acknowledgments` operation will block until the `DataWriter` has retransmitted its entire history, which is therefore effectively empty, or until the `max_wait` timeout expires, whichever happens first. In the first case the operation will return `RETCODE_OK`, in the latter it will return `RETCODE_TIMEOUT`.

⚠ Be aware that in case the operation returns `RETCODE_OK`, the data has only been acknowledged by the local infrastructure: it does not mean all remote subscriptions have already received the data. However, delivering the data to remote nodes is then the sole responsibility of the networking service: even when the publishing application would terminate, all data that has not yet been received may be considered 'on-route' and will therefore eventually arrive (unless the networking service itself will crash). In contrast, if the `DataWriter` would still have data in its local history buffer when it terminates, this data is considered 'lost'.

This operation is intended to be used only if the `DataWriter` has its `ReliabilityQosPolicyKind` set to `RELIABLE_RELIABILITY_QOS`. Otherwise the operation will return immediately with `RETCODE_OK`, since best-effort DataWriters will never store rejected samples in their local history: they will just drop them and continue business as usual.

Return Code

When the operation returns:

- *RETCODE_OK* - the data of the `DataWriter` has been acknowledged by the local infrastructure.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataWriter` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `DataWriter` is not enabled.

- *RETCODE_TIMEOUT* - not all data is acknowledged before `max_wait` elapsed.

3.4.2.27 write (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
   write
      (<data> instance_data,
        long handle);
```

3.4.2.28 write_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
```

```
write_w_timestamp
  (<data> instance_data,
    int handle,
    Time_t source_timestamp);
```

3.4.2.29 writedispose (abstract)

This abstract operation is defined as a generic operation, which is implemented by
the `<type>DataWriter` class. Therefore, to use this operation, the data type
specific implementation of this operation in its respective derived class must be
used. For further explanation see the description for the fictional data type `Foo`
derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
  writedispose
    (<data> instance_data,
      long handle);
```

3.4.2.30 writedispose_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by
the `<type>DataWriter` class. Therefore, to use this operation, the data type
specific implementation of this operation in its respective derived class must be
used. For further explanation see the description for the fictional data type `Foo`
derived `FooDataWriter` class.

Synopsis

```
import DDS.*;
public int
  writedispose
    (<data> instance_data,
      long handle,
      Time_t source_timestamp);
```

3.4.2.31 Interface FooDataWriter

The pre-processor generates from IDL type descriptions the application
`<type>DataWriter` interfaces. For each application data type that is used as
`Topic` data type, a typed interface `<type>DataWriter` is extending the
`DataWriter` interface. In this paragraph, the interface `FooDataWriter` in the
package `SPACE` describes the operations of these extending `<type>DataWriter`
interface as an example for the fictional application type `Foo` (defined in the
module `SPACE`).

A `FooDataWriter` is attached to exactly one `Publisher` which acts as a factory for it. The `FooDataWriter` is bound to exactly one `Topic` that has been registered to use a data type `Foo`. The `Topic` must exist prior to the `FooDataWriter` creation.

The interface description is as follows:

```
 public interface FooDataWriter
{
//
// extends interface Entity
//
// public StatusCondition
//     get_statuscondition
//         (void);
// public int
//     get_status_changes
//         (void);
// public int
//     enable
//         (void);
//
// inherited from class DataWriter
//
// public int
//     set_qos
//         (DataWriterQos qos);

// public int
//     get_qos
//         (DataWriterQosHolder qos);

// public int
//     set_listener
//         (DataWriterListener a_listener,
//         int mask);

// public DataWriterListener
//     get_listener
//         (void);

// public Topic
//     get_topic
//         (void);

// public Publisher
//     get_publisher
//         (void);

// public int
```

PRISMTECH

```
//       wait_for_acknowledgments
//           (Duration_t max_wait);

// public int
//     get_liveliness_lost_status
//         (LivelinessLostStatusHolder status);

// public int
//     get_offered_deadline_missed_status
//         (OfferedDeadlineMissedStatusHolder status);

// public int
//     get_offered_incompatible_qos_status
//         (OfferedIncompatibleQosStatusHolder status);

// public int
//     get_publication_matched_status
//         (PublicationMatchedStatusHolder status);

// public int
//     assert_liveliness
//         (void);

// public int
//     get_matched_subscriptions
//         (InstanceHandleSeqHolder subscription_handles);

// public int
//     get_matched_subscription_data
//         (SubscriptionBuiltinTopicDataHolder subscription_data,
//         long subscription_handle);
//
// implemented API operations
//
    public long
        register_instance
            (Foo instance_data);
    public long
        register_instance_w_timestamp
            (Foo instance_data,
               Time_t source_timestamp);
    public int
        unregister_instance
            (Foo instance_data,
               long handle);
    public int
        unregister_instance_w_timestamp
            (Foo instance_data,
               long handle,
               Time_t source_timestamp);
```

```
            public int
               write
                  (Foo instance_data,
                    long handle);
            public int
               write_w_timestamp
                  (Foo instance_data,
                    long handle,
                    Time_t source_timestamp);
            public int
               dispose
                  (Foo instance_data,
                    long instance_handle);
            public int
               dispose_w_timestamp
                  (Foo instance_data,
                    long instance_handle,
                    Time_t source_timestamp);
            public int
               writedispose
                  (Foo instance_data,
                    long instance_handle);
            public int
               writedispose_w_timestamp
                  (Foo instance_data,
                    long instance_handle,
                    Time_t source_timestamp);
            public int
               get_key_value
                  (FooHolder key_holder,
                    long handle);
            public long
               lookup_instance
                  (Foo instance_data);
         };
```

The following paragraphs describe the usage of all `FooDataWriter` operations.
The inherited operations are listed but not fully described because they are not
implemented in this class. The full description of these operations is given in the
classes from which they are inherited.

3.4.2.32 **assert_liveliness (inherited)**

This operation is inherited and therefore not described here. See the interface
`DataWriter` for further explanation.

Synopsis

```
         public int
            assert_liveliness
```

```
                                 (void);
```

3.4.2.33 dispose

Scope

```
    SPACE.FooDataWriter
```

Synopsis

```
public int
    dispose
      (Foo instance_data,
        long instance_handle);
```

Description

This operation requests the Data Distribution Service to mark the instance for deletion.

Parameters

in Foo instance_data - the actual instance to be disposed of.

in long instance_handle - the handle to the instance to be disposed of.

Return Value

int - Possible return codes of the operation are:
```
    RETCODE_OK,  RETCODE_ERROR,  RETCODE_BAD_PARAMETER,
    RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,
    RETCODE_NOT_ENABLED RETCODE_PRECONDITION_NOT_MET or
    RETCODE_TIMEOUT.
```

Detailed Description

This operation requests the Data Distribution Service to mark the instance for deletion. Copies of the instance and its corresponding samples, which are stored in every connected `DataReader` and, dependent on the `QoSPolicy` settings, also in the Transient and Persistent stores, will be marked for deletion by setting their `InstanceStateKind` to `NOT_ALIVE_DISPOSED_INSTANCE_ STATE`.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

As a side effect, this operation asserts liveliness on the `DataWriter` itself and on the containing `DomainParticipant`.

Effects on DataReaders

Actual deletion of the instance administration in a connected `DataReader` will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it.

  - A `DataWriter` can register an instance explicitly by using one of the special operations `register_instance` or `register_instance_w_timestamp`.

  - A `DataWriter` can register an instance implicitly by using the special constant `HANDLE_NIL` in any of the other `DataWriter` operations.

  - A `DataWriter` can unregister an instance explicitly by using one of the special operations `unregister_instance` or `unregister_instance_w_timestamp`.

  - A `DataWriter` will unregister all its contained instances implicitly when it is deleted.

  - When a `DataReader` detects a loss of liveliness in one of its connected `DataWriters`, it will consider all instances registered by that `DataWriter` as being implicitly unregistered.

- **and** the application must have consumed all samples belonging to the instance, either implicitly or explicitly.

  - An application can consume samples explicitly by invoking the `take` operation, or one of its variants, on its `DataReaders`.

  - The `DataReader` can consume disposed samples implicitly when the `autopurge_disposed_samples_delay` of the `ReaderData LifecycleQosPolicy` has expired.

The `DataReader` may also remove instances that haven't been disposed first: this happens when the `autopurge_nowriter_samples_delay` of the `ReaderDataLifecycleQosPolicy` has expired after the instance is considered unregistered by all connected `DataWriters` (i.e. when it has a `InstanceStateKind` of `NOT_ALIVE_NO_WRITERS`). See also Section 3.1.3.15, *ReaderDataLifecycleQosPolicy*, on page 76.

Effects on Transient/Persistent Stores

Actual deletion of the instance administration in the connected Transient and Persistent stores will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it. (See above.)

- **_and_** the period of time specified by the `service_cleanup_delay` attribute in the `DurabilityServiceQosPolicy` on the `Topic` must have elapsed after the instance is considered unregistered by all connected `DataWriters`.

See also Section 3.1.3.4, *DurabilityServiceQosPolicy*, on page 53.

Instance Handle

The `HANDLE_NIL constant` can be used for the parameter `instance_handle`. This indicates the identity of the `instance` is automatically deduced from the `instance_data` (by means of the `key`).

If `instance_handle` is any value other than `HANDLE_NIL`, it must correspond to the value that was returned by either the `register_instance` operation or the `register_instance_w_timestamp` operation, when the instance (identified by its `key`) was registered. If there is no correspondence, the result of the operation is unspecified.

The sample that is passed as `instance_data` is only used to check for consistency between its `key` values and the supplied `instance_handle`: the sample itself will not actually be delivered to the connected `DataReaders`. Use the `writedispose` operation if the sample itself should be delivered together with the dispose request.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `dispose` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `dispose` operation may block (waiting for space to become available). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits, the `SPACE_FooDataWriter_dispose` operation will fail and returns `RETCODE_TIMEOUT`.

Sample Validation

Since the sample that is passed as `instance_data` is merely used to check for consistency between its `key` values and the supplied `instance_handle`, only these keyfields will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- a string (bounded or unbounded) may not be `null`. (Use "" for an empty string instead)

- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated, the operation will fail and return a RETCODE_BAD_PARAMETER. More specific information about the context of this error will be written to the error log

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service is informed that the instance data must be disposed of
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - instance_handle is not a valid handle or instance_data is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the FooDataWriter has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the FooDataWriter is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the instance_handle has not been registered with this FooDataWriter.
- *RETCODE_TIMEOUT* - the current action overflowed the available resources as specified by the combination of the ReliabilityQosPolicy, HistoryQosPolicy and ResourceLimitsQosPolicy. This caused blocking of the dispose operation, which could not be resolved before max_blocking_time of the ReliabilityQosPolicy elapsed.

3.4.2.34 **dispose_w_timestamp**

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
   dispose_w_timestamp
      (Foo instance_data,
        long instance_handle,
        Time_t source_timestamp);
```

Description

This operation requests the Data Distribution Service to mark the instance for deletion and provides a value for the source_timestamp explicitly.

Parameters

in Foo instance_data - the actual instance to be disposed of.

in long instance_handle - the handle to the instance to be disposed of.

in Time_t source_timestamp - the timestamp which is provided for the
`DataReader`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,`
`RETCODE_ALREADY_ DELETED, RETCODE_OUT_OF_RESOURCES,`
`RETCODE_NOT_ENABLED RETCODE_PRECONDITION_NOT_MET` or
`RETCODE_TIMEOUT.`

Detailed Description

This operation performs the same functions as `dispose` except that the application
provides the value for the `source_timestamp` that is made available to connected
`DataReader` objects. This timestamp is important for the interpretation of the
`DestinationOrderQosPolicy`.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service is informed that the instance data
  must be disposed of

- *RETCODE_ERROR* - an internal error has occurred

- *RETCODE_BAD_PARAMETER* - `instance_handle` is not a valid handle or
  `instance_data` is not a valid sample.

- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - the `instance_handle` has not been
  registered with this `FooDataWriter`.

- *RETCODE_TIMEOUT* - the current action overflowed the available resources as
  specified by the combination of the `ReliabilityQosPolicy`,
  `HistoryQosPolicy` and `ResourceLimitsQosPolicy`. This caused blocking
  of the `dispose_w_timestamp` operation, which could not be resolved before
  `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.35 **enable (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
int
    enable
        (void);
```

3.4.2.36 **get_key_value**

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
int
    get_key_value
        (FooHolder key_holder,
         long handle);
```

Description

This operation retrieves the key value of a specific instance.

Parameters

inout FooHolder key_holder - a reference to the sample in which the key values are stored.

in long handle - the handle to the instance from which to get the key value.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation retrieves the key value of the instance referenced to by `instance_handle`. When the operation is called with an `HANDLE_NIL` constant as an `instance_handle`, the operation will return `RETCODE_BAD_PARAMETER`. The operation will only fill the fields that form the key inside the `key_holder` instance. This means that the non-key fields are not applicable and may contain garbage.

The operation must only be called on registered instances. Otherwise the operation returns the error `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `key_holder` instance contains the key values of the instance;
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - `handle` is not a valid handle
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - this instance is not registered.

3.4.2.37 get_listener (inherited)

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public DataWriterListener
   get_listener
      (void);
```

3.4.2.38 get_liveliness_lost_status (inherited)

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_liveliness_lost_status
      (LivelinessLostStatusHolder status);
```

3.4.2.39 get_matched_subscription_data (inherited)

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
```

```
get_matched_subscription_data
    (SubscriptionBuiltinTopicData subscription_data,
     long subscription_handle);
```

3.4.2.40 get_matched_subscriptions (inherited)

This operation is inherited and therefore not described here. See the interface
`DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_matched_subscriptions
        (InstanceHandleSeqHolder subscription_handles);
```

3.4.2.41 get_offered_deadline_missed_status (inherited)

This operation is inherited and therefore not described here. See the interface
`DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_offered_deadline_missed_status
        (OfferedDeadlineMissedStatusHolder status);
```

3.4.2.42 get_offered_incompatible_qos_status (inherited)

This operation is inherited and therefore not described here. See the interface
`DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_offered_incompatible_qos_status
        (OfferedIncompatibleQosStatusHolder status);
```

3.4.2.43 get_publication_matched_status (inherited)

This operation is inherited and therefore not described here. See the interface
`DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_publication_matched_status
        (PublicationMatchedStatusHolder status);
```

3.4.2.44 **get_publisher (inherited)**

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public Publisher
   get_publisher
      (void);
```

3.4.2.45 **get_qos (inherited)**

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_qos
      (DataWriterQosHolder qos);
```

3.4.2.46 **get_status_changes (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
      (void);
```

3.4.2.47 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
      (void);
```

3.4.2.48 **get_topic (inherited)**

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public Topic
   get_topic
       (void);
```

3.4.2.49 lookup_instance

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public long
   lookup_instance
       (Foo instance_data);
```

Description

This operation returns the value of the instance handle which corresponds to the `instance_data`.

Parameters

in Foo instance_data - a reference to the instance for which the corresponding instance handle needs to be looked up.

Return Value

long - Result value is the instance handle which corresponds to the `instance_data`.

Detailed Description

This operation returns the value of the instance handle which corresponds to the `instance_data`. The `instance_data` parameter is only used for the purpose of examining the fields that define the `key`. The instance handle can be used in any `write`, `dispose` or `unregister` operations (or their timestamped variants) that operate on a specific instance. Note that `DataWriter` instance handles are local, and are not interchangeable with `DataReader` instance handles nor with instance handles of an other `DataWriter`.

This operation does not register the instance in question. If the instance has not been previously registered, if the `DataWriter` is already deleted or if for any other reason the Service is unable to provide an instance handle, the Service will return the special value `HANDLE_NIL`.

Sample Validation

Since the sample that is passed as `instance_data` is merely used to determine the identity based on the uniqueness of its `key` values, only the keyfields will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- a string (bounded or unbounded) may not be `null`. (Use "" for an empty string instead)

- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated, the operation will fail and return a `HANDLE_NIL`. More specific information about the context of this error will be written to the error log.

3.4.2.50 **register_instance**

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
long
    register_instance
        (Foo instance_data);
```

Description

This operation informs the Data Distribution Service that the application will be modifying a particular instance.

Parameters

in Foo instance_data - the instance, which the application writes to or disposes of.

Return Value

long - Result value is the handle to the Instance, which may be used for writing and disposing of. In case of an error, a `HANDLE_NIL constant` is returned.

Detailed Description

This operation informs the Data Distribution Service that the application will be modifying a particular instance. This operation may be invoked prior to calling any operation that modifies the instance, such as `write`, `write_w_timestamp`, `unregister_instance`, `unregister_instance_w_timestamp`, `dispose`, `dispose_w_timestamp`, `writedispose` and `writedispose_w_timestamp`. When the application does register the instance before modifying, the Data

Distribution Service will handle the instance more efficiently. It takes as a parameter (`instance_data`) an instance (to get the `key` value) and returns a handle that can be used in successive `DataWriter` operations. In case of an error, a `HANDLE_NIL` is returned.

The explicit use of this operation is optional as the application can directly call the `write`, `write_w_timestamp`, `unregister_instance`, `unregister_instance_w_timestamp`, `dispose`, `dispose_w_timestamp`, `writedispose` and `writedispose_w_timestamp` operations and specify a `HANDLE_NIL` value to indicate that the `sample` should be examined to identify the instance.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `register_instance` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. In case the `synchronous` attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `register_instance` operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the `register_instance` operation will fail and returns `HANDLE_NIL`.

Sample Validation

Since the sample that is passed as `instance_data` is merely used to determine the identity based on the uniqueness of its `key` values, only the keyfields will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- a string (bounded or unbounded) may not be `null`. (Use `""` for an empty string instead)

- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated, the operation will fail and return a
`HANDLE_NIL`. More specific information about the context of this error will be
written to the error log.

Multiple Calls

If this operation is called for an already registered instance, it just returns the already
allocated instance handle. This may be used to look up and retrieve the handle
allocated to a given instance.

3.4.2.51 register_instance_w_timestamp

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public long
   register_instance_w_timestamp
      (Foo instance_data,
        Time_t source_timestamp);
```

Description

This operation will inform the Data Distribution Service that the application will be
modifying a particular instance and provides a value for the `source_timestamp`
explicitly.

Parameters

in Foo instance_data - the instance, which the application will write to or
dispose of.

in Time_t source_timestamp - the timestamp used.

Return Value

long - Result value is the handle to the `Instance`, which must be used for writing
and disposing. In case of an error, a `HANDLE_NIL` reference is returned.

Detailed Description

This operation performs the same functions as `register_instance` except that
the application provides the value for the `source_timestamp` that is made
available to connected `DataReader` objects. This timestamp is important for the
interpretation of the `DestinationOrderQosPolicy`.

Multiple Calls

If this operation is called for an already registered instance, it just returns the already allocated instance handle. The source_timestamp is ignored in that case.

3.4.2.52 set_listener (inherited)

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
   set_listener
      (DataWriterListener a_listener,
         int mask);
```

3.4.2.53 set_qos (inherited)

This operation is inherited and therefore not described here. See the interface `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
   set_qos
      (DataWriterQos qos);
```

3.4.2.54 unregister_instance

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
   unregister_instance
      (Foo instance_data,
         long handle);
```

Description

This operation informs the Data Distribution Service that the application will **not** be modifying a particular instance any more.

Parameters

in Foo instance_data - the instance to which the application was writing or disposing.

PRISMTECH

in long handle - the handle to the `Instance`, which has been used for writing and disposing.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation informs the Data Distribution Service that the application will **not** be modifying a particular instance any more. Therefore, this operation reverses the action of `register_instance` or `register_instance_w_timestamp`. It should only be called on an `instance` that is currently registered. This operation should be called just once per `instance`, regardless of how many times `register_instance` was called for that `instance`. This operation also indicates that the Data Distribution Service can locally remove all information regarding that `instance`. The application should not attempt to use the `handle`, previously allocated to that `instance`, after calling this operation.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Effects

If, after unregistering, the application wants to modify (`write` or `dispose`) the `instance`, it has to `register` the instance again, or it has to use the special constant `HANDLE_NIL`.

This operation does not indicate that the `instance` should be deleted (that is the purpose of `dispose`). This operation just indicates that the `DataWriter` no longer has "anything to say" about the `instance`. If there is no other `DataWriter` that has registered the instance as well, then the `InstanceStateKind` in all connected `DataReaders` will be changed to `NOT_ALIVE_NO_WRITERS_ INSTANCE_STATE`, provided this `InstanceStateKind` was not already set to `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. In the last case the `InstanceStateKind` will not be effected by the `unregister_instance` call, see also Figure 21, *State Chart of the instance_state for a Single Instance*, on page 574.

This operation can affect the `ownership` of the data `instance`. If the `DataWriter` was the exclusive owner of the `instance`, calling this operation will release that `ownership`, meaning `ownership` may be transferred to another, possibly lower strength, `DataWriter`.

The operation must be called only on registered instances. Otherwise the operation returns the error `RETCODE_PRECONDITION_NOT_MET`.

Instance Handle

The special constant `HANDLE_NIL` can be used for the parameter `handle`. This indicates that the identity of the `instance` is automatically deduced from the `instance_data` (by means of the `key`).

If `handle` is any value other than the special constant `HANDLE_NIL`, then it must correspond to the value returned by `register_instance` or `register_instance_w_timestamp` when the `instance` (identified by its `key`) was registered. If there is no correspondence , the result of the operation is unspecified.

The sample that is passed as `instance_data` is only used to check for consistency between its `key` values and the supplied `instance_handle`: the sample itself will not actually be delivered to the connected `DataReaders`.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `unregister_instance` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. In case the `synchronous` attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `unregister_instance` operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the `unregister_instance` operation will fail and returns `HANDLE_NIL`.

Sample Validation

Since the sample that is passed as `instance_data` is merely used to check for consistency between its `key` values and the supplied `instance_handle`, only these keyfields will be validated against the restrictions imposed by the IDL to Java language mapping, where:

PRISMTECH

- a string (bounded or unbounded) may not be `null`. (Use `""` for an empty string instead)

- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated, the operation will fail and return a `RETCODE_BAD_PARAMETER`. More specific information about the context of this error will be written to the error log.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service is informed that the instance will not be modified any more

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - `handle` is not a valid handle or `instance_data` is not a valid sample.

- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted

- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - the `handle` has not been registered with this `FooDataWriter`.

- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the `ReliablityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `unregister_instance` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.55 unregister_instance_w_timestamp

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
   unregister_instance_w_timestamp
      (Foo instance_data,
        long handle,
        Time_t source_timestamp);
```

Description

This operation will inform the Data Distribution Service that the application will **not** be modifying a particular instance any more and provides a value for the source_timestamp explicitly.

Parameters

in `Foo instance_data` - the instance to which the application was writing or disposing.

in `long handle` - the handle to the `Instance`, which has been used for writing and disposing.

in `Time_t source_timestamp` - the timestamp used.

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_TIMEOUT.

Detailed Description

This operation performs the same functions as unregister_instance except that the application provides the value for the source_timestamp that is made available to connected DataReader objects. This timestamp is important for the interpretation of the DestinationOrderQosPolicy.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service is informed that the instance will not be modified any more
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - handle is not a valid handle or instance_data is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the FooDataWriter has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the FooDataWriter is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this FooDataWriter.

PRISMTECH

- *RETCODE_TIMEOUT* - the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`. This caused blocking of the `unregister_instance_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the `ReliablityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `unregister_instance_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.56 wait_for_acknowledgments (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
import DDS.*;
public int
   wait_for_acknowledgments
      (const Duration_t& max_wait);
```

3.4.2.57 write

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
   write
      (Foo instance_data,
        long handle);
```

Description

This operation modifies the value of a data instance.

Parameters

in Foo instance_data - the data to be written.

in long handle - the handle to the instance as supplied by register_instance.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,`
`RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,`
`RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or
`RETCODE_TIMEOUT.`

Detailed Description

This operation modifies the value of a data instance. When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

As a side effect, this operation asserts liveliness on the `DataWriter` itself and on the containing `DomainParticipant`.

Before writing data to an `instance`, the instance may be registered with the `register_instance` or `register_instance_w_timestamp` operation. The `handle` returned by one of the `register_instance` operations can be supplied to the parameter `handle` of the `write` operation. However, it is also possible to supply the special constant `HANDLE_NIL`, which means that the identity of the `instance` is automatically deduced from the `instance_data` (identified by the `key`).

Instance Handle

The special constant `HANDLE_NIL` can be used for the parameter `handle`. This indicates the identity of the `instance` is automatically deduced from the `instance_data` (by means of the `key`).

If `handle` is any value other than the special constant `HANDLE_NIL`, it must correspond to the value returned by `register_instance` or `register_instance_w_timestamp` when the `instance` (identified by its `key`) was registered. Passing such a registered `handle` helps the Data Distribution Service to process the sample more efficiently. If there is no correspondence between handle and sample, the result of the operation is unspecified.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `write` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, is exceeded. In case the synchronous attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous

PRISMTECH

DataReaders have acknowledged the data. Under these circumstances, the max_blocking_time attribute of the ReliabilityQosPolicy configures the maximum time the write operation may block (either waiting for space to become available or data to be acknowledged). If max_blocking_time elapses before the DataWriter is able to store the modification without exceeding the limits and all expected acknowledgements are received, the write operation will fail and returns RETCODE_TIMEOUT.

Sample Validation

Before the sample is accepted by the DataWriter, it will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- a string (bounded or unbounded) may not be null. (Use "" for an empty string instead)
- the length of a bounded string may not exceed the limit specified in IDL
- the length of a bounded sequence may not exceed the limit specified in IDL
- the length of an array must exactly match the size specified in IDL

If any of these restrictions is violated, the operation will fail and return a RETCODE_BAD_PARAMETER. More specific information about the context of this error will be written to the error log.

Return Code

When the operation returns:

- *RETCODE_OK* - the value of a data instance is modified
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - handle is not a valid handle or instance_data is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the FooDataWriter has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the FooDataWriter is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this FooDataWriter.
- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the ReliabilityQosPolicy, HistoryQosPolicy and ResourceLimitsQosPolicy, or the current action was waiting for data delivery acknowledgement by synchronous DataReaders. This caused blocking of the write operation, which could not be resolved before max_blocking_time of the ReliabilityQosPolicy elapsed.

3.4.2.58 **write_w_timestamp**

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
   write_w_timestamp
     (Foo instance_data,
       long handle,
       Time_t source_timestamp);
```

Description

This operation modifies the value of a data instance and provides a value for the `source_timestamp` explicitly.

Parameters

in Foo instance_data - the data to be written.

in long handle - the handle to the instance as supplied by `register_instance`.

in Time_t source_timestamp - the timestamp used.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,`
   `RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,`
   `RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or
   `RETCODE_TIMEOUT.`

Detailed Description

This operation performs the same functions as `write` except that the application provides the value for the parameter `source_timestamp` that is made available to `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Return Code

When the operation returns:

• *RETCODE_OK* - the value of a data instance is modified

• *RETCODE_ERROR* - an internal error has occurred

- *RETCODE_BAD_PARAMETER* - handle is not a valid handle or instance_data is not a valid sample.

- *RETCODE_ALREADY_DELETED* - the FooDataWriter has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the FooDataWriter is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this FooDataWriter.

- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the ReliabilityQosPolicy, HistoryQosPolicy and ResourceLimitsQosPolicy, or the current action was waiting for data delivery acknowledgement by synchronous DataReaders. This caused blocking of the write_w_timestamp operation, which could not be resolved before max_blocking_time of the ReliabilityQosPolicy elapsed.

3.4.2.59 writedispose

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
   writedispose
       (Foo instance_data,
        long handle);
```

Description

This operation modifies and disposes a data instance.

Parameters

in Foo instance_data - the data to be written and disposed.

in long handle - the handle to the instance as supplied by register_instance.

Return Value

`int` - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`,
`RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`,
`RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or
`RETCODE_TIMEOUT`.

Detailed Description

This operation requests the Data Distribution Service to modify the instance and mark it for deletion. Copies of the instance and its corresponding samples, which are stored in every connected `DataReader` and, dependent on the `QoSPolicy` settings, also in the Transient and Persistent stores, will be modified and marked for deletion by setting their `InstanceStateKind` to `NOT_ALIVE_DISPOSED_INSTANCE_STATE`.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

As a side effect, this operation asserts liveliness on the `DataWriter` itself and on the containing `DomainParticipant`.

Effects on DataReaders

Actual deletion of the instance administration in a connected `DataReader` will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it.
  - A `DataWriter` can register an instance explicitly by using one of the special operations `register_instance` or `register_instance_w_timestamp`.
  - A `DataWriter` can register an instance implicitly by using the special constant `HANDLE_NIL` in any of the other `DataWriter` operations.
  - A `DataWriter` can unregister an instance explicitly by using one of the special operations `unregister_instance` or `unregister_instance_w_timestamp`.
  - A `DataWriter` will unregister all its contained instances implicitly when it is deleted.
  - When a `DataReader` detects a loss of liveliness in one of its connected `DataWriters`, it will consider all instances registered by that `DataWriter` as being implicitly unregistered.

- **and** the application must have consumed all samples belonging to the instance, either implicitly or explicitly.

  - An application can consume samples explicitly by invoking the `take` operation, or one of its variants, on its `DataReaders`.
  - The `DataReader` can consume disposed samples implicitly when the `autopurge_disposed_samples_delay` of the `ReaderData LifecycleQosPolicy` has expired.

The `DataReader` may also remove instances that haven't been disposed first: this happens when the `autopurge_nowriter_samples_delay` of the `ReaderDataLifecycleQosPolicy` has expired after the instance is considered unregistered by all connected `DataWriters` (i.e. when it has a `InstanceStateKind` of `NOT_ALIVE_NO_WRITERS`). See also Section 3.1.3.15, *ReaderDataLifecycleQosPolicy*, on page 76.

Effects on Transient/Persistent Stores

Actual deletion of the instance administration in the connected Transient and Persistent stores will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it. (See above.)
- **and** the period of time specified by the `service_cleanup_delay` attribute in the `DurabilityServiceQosPolicy` on the `Topic` must have elapsed after the instance is considered unregistered by all connected `DataWriters`.

See also Section 3.1.3.4, *DurabilityServiceQosPolicy*, on page 53.

Instance Handle

The `HANDLE_NIL` handle value can be used for the parameter `handle`. This indicates the identity of the `instance` is automatically deduced from the `instance_data` (by means of the `key`).

If `handle` is any value other than `HANDLE_NIL`, it must correspond to the value that was returned by either the `register_instance` operation or the `register_instance_w_timestamp` operation, when the instance (identified by its `key`) was registered. If there is no correspondence, the result of the operation is unspecified.

The sample that is passed as `instance_data` will actually be delivered to the connected `DataReaders`, but will immediately be marked for deletion.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `writedispose` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. In case the `synchronous` attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `writedispose` operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the `writedispose` operation will fail and returns `RETCODE_TIMEOUT`.

Sample Validation

Before the sample is accepted by the DataWriter, it is validated against the restrictions imposed by the IDL to Java language mapping, where:

- a string (bounded or unbounded) may not be `null`. (Use "" for an empty string instead)

- the length of a bounded string may not exceed the limit specified in IDL

- the length of a bounded sequence may not exceed the limit specified in IDL

- the length of an array must exactly match the size specified in IDL

If any of these restrictions is violated, the operation will fail and return a `RETCODE_BAD_PARAMETER`. More specific information about the context of this error will be written to the error log.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service has modified the instance and marked it for deletion.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - `handle` is not a valid handle or `instance_data` is not a valid sample.

- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

PRISMTECH

- *RETCODE_NOT_ENABLED* - the FooDataWriter is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this SPACE_FooDataWriter.
- *RETCODE_TIMEOUT* - the current action overflowed the available resources as specified by the combination of the ReliabilityQosPolicy, HistoryQosPolicy and ResourceLimitsQosPolicy. This caused blocking of the writedispose operation, which could not be resolved before max_blocking_time of the ReliabilityQosPolicy elapsed.

3.4.2.60 writedispose_w_timestamp

Scope

```
SPACE.FooDataWriter
```

Synopsis

```
import DDS.*;
public int
    writedispose_w_timestamp
        (Foo instance_data,
         long handle,
         Time_t source_timestamp);
```

Description

This operation requests the Data Distribution Service to modify the instance and mark it for deletion, and provides a value for the source_timestamp explicitly.

Parameters

in Foo instance_data - the data to be written and disposed.

in long handle - the handle to the instance as supplied by register_instance.

in Time_t source_timestamp - the timestamp used.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_TIMEOUT.

Detailed Description

This operation performs the same functions as `writedispose` except that the application provides the value for the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service has modified the instance and marked it for deletion.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - `handle` is not a valid handle or `instance_data` is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the `handle` has not been registered with this `SPACE_FooDataWriter`.
- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `writedispose_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.3 **PublisherListener interface**

Since a `Publisher` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `PublisherListener`. This interface must be implemented by the application. A user-defined class must be provided by the application which must extend from the `PublisherListener` class. **All** `PublisherListener` operations **must** be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

⚠ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

PRISMTECH

The `PublisherListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `PublisherListener` is related to changes in communication status.

The interface description is as follows:

```
 public interface PublisherListener {
//
// extends interface DataWriterListener
//
// public void
//     on_offered_deadline_missed
//         (DataWriter writer,
//         OfferedDeadlineMissedStatus status);

// public void
//     on_offered_incompatible_qos
//         (DataWriter writer,
//         OfferedIncompatibleQosStatus status);

// public void
//     on_liveliness_lost
//         (DataWriter writer,
//         LivelinessLostStatus status);

// public void
//     on_publication_matched
//         (DataWriter writer,
//         PublicationMatchedStatus status);
//
// implemented API operations
//        <no operations>
//
};
```

The following paragraphs list all `PublisherListener` operations. Since these operations are all inherited, they are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.4.3.1 **on_liveliness_lost (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
public void
```

```
on_liveliness_lost
   (DataWriter writer,
      LivelinessLostStatus status);
```

3.4.3.2 **on_offered_deadline_missed (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_offered_deadline_missed
      (DataWriter writer,
         OfferedDeadlineMissedStatus status);
```

3.4.3.3 **on_offered_incompatible_qos (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_offered_incompatible_qos
      (DataWriter writer,
         OfferedIncompatibleQosStatus status);
```

3.4.3.4 **on_publication_matched (inherited, abstract)**

This operation is inherited and therefore not described here. See the interface `DataWriterListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_publication_matched
      (DataWriter writer,
         PublicationMatchedStatus status);
```

3.4.4 **DataWriterListener interface**

Since a `DataWriter` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `DataWriterListener`. This interface must be implemented by the application. A user-defined class must be provided by the application which must implement the

DataWriterListener interface. **All** DataWriterListener operations **must** be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

⚠️ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The DataWriterListener provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a QosPolicy setting, etc. The DataWriterListener is related to changes in communication status.

The interface description is as follows:

```
 public interface DataWriterListener
 {
 //
 // operations
 //
    public void
       on_offered_deadline_missed
          (DataWriter writer,
          OfferedDeadlineMissedStatus status);

    public void
       on_offered_incompatible_qos
          (DataWriter writer,
          OfferedIncompatibleQosStatus status);

    public void
       on_liveliness_lost
          (DataWriter writer,
          LivelinessLostStatus status);

    public void
       on_publication_matched
          (DataWriter writer,
          PublicationMatchedStatus status);
 //
 // implemented API operations
 //      <no operations>
 //
 };
```

The DataWriterListener abstract operations are fully described because they must be implemented by the application.

3.4.4.1 **on_liveliness_lost**

Scope

```
DDS.DataWriterListener
```

Synopsis

```
import DDS.*;
public void
   on_liveliness_lost
      (DataWriter writer,
        LivelinessLostStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `LivelinessLostStatus` changes.

Parameters

in DataWriter writer - contains a reference to the `DataWriter` on which the `LivelinessLostStatus` has changed (this is an input to the application).

in LivelinessLostStatus status - contain the `LivelinessLostStatus` object (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `LivelinessLostStatus` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataWriterListener` is installed and enabled for the liveliness lost status. The liveliness lost status will change when the liveliness that the `DataWriter` has committed through its `LivelinessQosPolicy` was not respected. In other words, the `DataWriter` failed to actively signal its liveliness within the offered liveliness period. As a result, the `DataReader` objects will consider the `DataWriter` as no longer "alive".

The Data Distribution Service will call the `DataWriterListener` operation with a parameter `writer`, which will contain a reference to the `DataWriter` on which the conflict occurred and a parameter `status`, which will contain the `LivelinessLostStatus` object.

3.4.4.2 **on_offered_deadline_missed**

Scope

```
DDS.DataWriterListener
```

Synopsis

```
import DDS.*;
public void
   on_offered_deadline_missed
      (DataWriter writer,
        OfferedDeadlineMissedStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `OfferedDeadlineMissedStatus` changes.

Parameters

in DataWriter writer - contain a reference to the `DataWriter` on which the `OfferedDeadlineMissedStatus` has changed (this is an input to the application).

in OfferedDeadlineMissedStatus status - contain the `OfferedDeadlineMissedStatus` object (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `OfferedDeadlineMissedStatus` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataWriterListener` is installed and enabled for the offered deadline missed status. The offered deadline missed status will change when the deadline that the `DataWriter` has committed through its `DeadlineQosPolicy` was not respected for a specific instance.

The Data Distribution Service will call the `DataWriterListener` operation with a parameter `writer`, which will contain a reference to the `DataWriter` on which the conflict occurred and a parameter `status`, which will contain the `OfferedDeadlineMissedStatus` object.

3.4.4.3 **on_offered_incompatible_qos**

Scope

```
DDS.DataWriterListener
```

Synopsis

```
import DDS.*;
public void
    on_offered_incompatible_qos
        (DataWriter writer,
          OfferedIncompatibleQosStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the OFFERED_INCOMPATIBLE_QOS_STATUS changes.

Parameters

in DataWriter writer - contain a reference to the DataWriter on which the OFFERED_INCOMPATIBLE_QOS_STATUS has changed (this is an input to the application).

in OfferedIncompatibleQosStatus status - contain the OfferedIncompatibleQosStatus object (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the OFFERED_INCOMPATIBLE_QOS_STATUS changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant DataWriterListener is installed and enabled for the OFFERED_INCOMPATIBLE_QOS_STATUS. The incompatible Qos status will change when a DataReader object has been discovered by the DataWriter with the same Topic and a requested DataReaderQos that was incompatible with the one offered by the DataWriter.

The Data Distribution Service will call the DataWriterListener operation with a parameter writer, which will contain a reference to the DataWriter on which the conflict occurred and a parameter status, which will contain the OfferedIncompatibleQosStatus object.

3.4.4.4 **on_publication_matched**

Scope

```
DDS.DataWriterListener
```

Synopsis

```
import DDS.*;
public void
   on_publication_matched
      (DataWriter writer,
         PublicationMatchedStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current publication, or when an existing match has ceased to exist.

Parameters

in DataWriter writer - contains a pointer to the `DataWriter` for which a match has been discovered (this is an input to the application provided by the Data Distribution Service).

in PublicationMatchedStatus status - contains the *PublicationMatchedStatus* object (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current publication, or when an existing match has ceased to exist. Usually this means that a new DataReader that matches the Topic and that has compatible Qos as the current `DataWriter` has either been discovered, or that a previously discovered DataReader has ceased to be matched to the current `DataWriter`. A DataReader may cease to match when it gets deleted, when it changes its Qos to a value that is incompatible with the current `DataWriter` or when either the `DataWriter` or the DataReader has chosen to put its matching counterpart on its ignore-list using the `ignore_subcription` or `ignore_publication` operations on the `DomainParticipant`.

The implementation of this Listener operation may be left empty when this functionality is not needed: it will only be called when the relevant `DataWriterListener` is installed and enabled for the `PUBLICATION_MATCHED_STATUS`.

The Data Distribution Service will provide a reference to the `DataWriter` in the parameter `writer` and the `PublicationMatchedStatus` object in the parameter `status` for use by the application.

3.5 **Subscription Module**
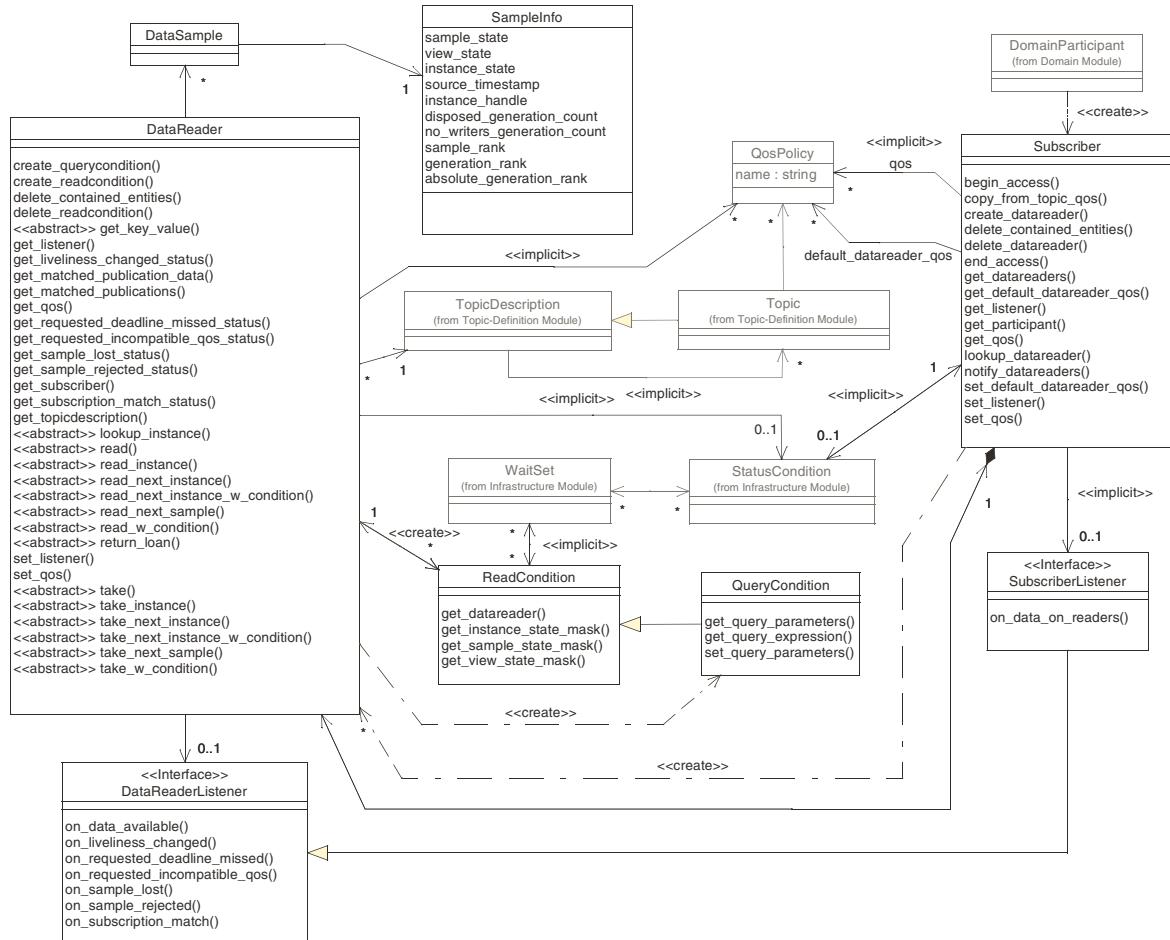
Figure 19 DCPS Subscription Module's Class Model

This module contains the following classes:

• `Subscriber`

- Subscription type specific classes
- `DataSample`
- `SampleInfo` (class)
- `SubscriberListener` (interface)
- `DataReaderListener` (interface)
- `ReadCondition`
- `QueryCondition`

"Subscription type specific classes" contains the generic class and the generated data type specific classes. For each data type, a data type specific class `<type>DataReader` is generated (based on IDL) by calling the pre-processor.

For instance, for the fictional data type `Foo` (this also applies to other types); "Subscription type specific classes" contains the following classes:

- `DataReader (abstract)`
- `FooDataReader`
- `DataReaderView (abstract)`
- `FooDataReaderView`

A `Subscriber` is an object responsible for receiving published data and making it available (according to the `SubscriberQos`) to the application. It may receive and dispatch `Topic` with data of different specified data types. To access the received data, the application must use a typed `DataReader` attached to the `Subscriber`. Thus, a subscription is defined by the association of a `DataReader` with a `Subscriber`. This association expresses the intent of the application to subscribe to the data described by the `DataReader` in the context provided by the `Subscriber`.

3.5.1 Interface Subscriber

A `Subscriber` is the object responsible for the actual reception of the data resulting from its subscriptions.

A `Subscriber` acts on behalf of one or more `DataReader` objects that are related to it. When it receives data (from the other parts of the system), it indicates to the application that data is available through its `DataReaderListener` and by enabling related `Conditions`. The application can access the list of concerned `DataReader` objects through the operation `get_datareaders` and then access the data available through operations on the `DataReader`.

The interface description of this class is as follows:

```
 public interface Subscriber
 {
 //
 // extends interface Entity
 //
```

```
                   // public StatusCondition
                   //    get_statuscondition
                   //        (void);
                   // public int
                   //    get_status_changes
                   //        (void);
                   // public int
                   //    enable
                   //        (void);
                   //
                   // implemented API operations
                   //
                      public DataReader
                         create_datareader
                            (TopicDescription a_topic,
                            DataReaderQos qos,
                            DataReaderListener a_listener,
                            int mask);

                      public int
                         delete_datareader
                            (DataReader a_datareader);

                      public int
                         delete_contained_entities
                            (void);

                      public DataReader
                         lookup_datareader
                            (String topic_name);

                      public int
                         get_datareaders
                            (DataReaderSeqHolder readers,
                            int sample_states,
                            int view_states,
                            int instance_states);

                      public int
                         notify_datareaders
                            (void);

                      public int
                         set_qos
                            (SubscriberQos qos);

                      public int
                         get_qos
                            (SubscriberQosHolder qos);
                      public int
```

```
                      set_listener
                         (SubscriberListener a_listener,
                           int mask);

                 public SubscriberListener
                      get_listener
                         (void);

                 public int
                      begin_access
                         (void);

                 public int
                      end_access
                         (void);

                 public DomainParticipant
                      get_participant
                         (void);

                 public int
                      set_default_datareader_qos
                         (DataReaderQos qos);

                 public int
                      get_default_datareader_qos
                         (DataReaderQosHolder qos);

                 public int
                      copy_from_topic_qos
                         (DataReaderQosHolder a_datareader_qos,
                          TopicQos a_topic_qos);
            };
```

The following paragraphs describe how all of the Subscriber operations are used. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.1.1 **begin_access**

Scope

```
    DDS.Subscriber
```

Synopsis

```
    import DDS.*;
    public int
        begin_access
            (void);
```

Description

This operation indicates that the application will begin accessing a coherent and/or ordered set of modifications that spans multiple DataReaders attached to this Subscriber. The access will be completed by a matching call to `end_access`.

Parameters

<none>

Return Value

int - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR or RETCODE_ALREADY_DELETED.

Detailed Description

This operation indicates that the application is about to access a set of coherent and/or ordered samples in any of the DataReader objects attached to the Subscriber. The operation will effectively lock all of the Subscriber's DataReader objects for any incoming modifications, so that the state of their history remains consistent for the duration of the access.

The application is required to use this operation only if the PresentationQosPolicy of the Subscriber to which the DataReader belongs has the `access_scope` set to 'GROUP'. In the aforementioned case, the operation `begin_access` must be called prior to calling any of the sample-accessing operations, namely: `get_datareaders` on the Subscriber and `read`, `take`, and all their variants on any DataReader. Otherwise the sample-accessing operations will return the error RETCODE_PRECONDITION_NOT_MET. Once the application has finished accessing the data samples it must call `end_access`.

It is not required for the application to call `begin_access/end_access` if the PresentationQosPolicy has the `access_scope` set to something other than 'GROUP'. Calling `begin_access/end_access` in this case is not considered an error and has no effect.

The calls to `begin_access/end_access` may be nested. In that case, the application must call `end_access` as many times as it called `begin_access`.

Return Code

When the operation returns:

- *RETCODE_OK* - access to coherent/ordered data has successfully started.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted.

3.5.1.2 **copy_from_topic_qos**

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
   copy_from_topic_qos
      (DataReaderQosHolder a_datareader_qos,
        TopicQos a_topic_qos);
```

Description

This operation will copy the policies in a_topic_qos to the corresponding policies in a_datareader_qos.

Parameters

inout DataReaderQosHolder a_datareader_qos - the destination DataReaderQos object to which the QosPolicy settings will be copied.

in TopicQos a_topic_qos - the source TopicQos, which will be copied.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation will copy the QosPolicy settings in a_topic_qos to the corresponding QosPolicy settings in a_datareader_qos (replacing the values in a_datareader_qos, if present).

This is a "convenience" operation, useful in combination with the operations get_default_datawriter_qos and Topic.get_qos. The operation copy_from_topic_qos can be used to merge the DataReader default QosPolicy settings with the corresponding ones on the Topic. The resulting DataReaderQos can then be used to create a new DataReader, or set its DataReaderQos.

This operation does not check the resulting a_datareader_qos for self consistency. This is because the "merged" a_datareader_qos may not be the final one, as the application can still modify some QosPolicy settings prior to applying the DataReaderQos to the DataReader.

When the operation returns:

- *RETCODE_OK* - the `QosPolicy` settings have successfully been copied from the `TopicQos` to the `DataReaderQos`

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.3 create_datareader

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public DataReader
   create_datareader
      (TopicDescription a_topic,
         DataReaderQos qos,
         DataReaderListener a_listener,
         int mask);
```

Description

This operation creates a `DataReader` with the desired `QosPolicy` settings, for the desired `TopicDescription` and attaches the optionally specified `DataWriterListener` to it.

Parameters

in TopicDescription a_topic - a reference to the `TopicDescription` for which the `DataReader` is created. This may be a `Topic`, `MultiTopic` or `ContentFilteredTopic`.

in DataReaderQos qos - the object with the `QosPolicy` settings for the new `DataReader`, when these `QosPolicy` settings are not self consistent, no `DataReader` is created.

in DataReaderListener a_listener - a reference to the `DataReaderListener` instance which will be attached to the new `DataReader` It is permitted to use `null` as the value of the listener: this behaves as a `DataWriterListener` whose operations perform no action.

in int mask - a bit-mask in which each bit enables the invocation of the `DataReaderListener` for a certain status.

Return Value

DataReader - Return value is a reference to the newly created `DataReader`. In
case of an error, the `null` reference is returned.

Detailed Description

This operation creates a `DataReader` with the desired `QosPolicy` settings, for the
desired `TopicDescription` and attaches the optionally specified
`DataReaderListener` to it. The `TopicDescription` may be a `Topic`,
`MultiTopic` or `ContentFilteredTopic`. The returned `DataReader` is attached
(and belongs) to the `Subscriber`. To delete the `DataReader` the operation
`delete_datareader` or `delete_contained_entities` must be used. If no
read rights are defined for the specific topic then the creation of the `DataReader`
will fail.

Application Data Type

The `DataReader` returned by this operation is an object of a derived class, specific
to the data type associated with the `TopicDescription`. For each
application-defined data type `<type>` there is a class `<type>DataReader`
generated by calling the pre-processor. This data type specific class extends
`DataReader` and contains the operations to read data of data type `<type>`.

Because the `DataReader` may read a `Topic`, `ContentFilteredTopic` or
`MultiTopic`, the `DataReader` is associated with the `TopicDescription`. The
`DataWriter` can only write a `Topic`, **not** a `ContentFilteredTopic` or
`MultiTopic`, because these two are constructed at the `Subscriber` side.

QosPolicy

The common application pattern to construct the `QosPolicy` settings for the
`DataReader` is to:

- Retrieve the `QosPolicy` settings on the associated `TopicDescription` by
  means of the `get_qos` operation on the `TopicDescription`
- Retrieve the default `DataReaderQos` by means of the
  `get_default_datareader_qos` operation on the `Subscriber`
- Combine those two `QosPolicy` settings and selectively modify policies as
  desired (`copy_from_topic_qos`)
- Use the resulting `QosPolicy` settings to construct the `DataReader`
- In case the specified `QosPolicy` settings are not self consistent, no `DataReader`
  is created and the `null` reference is returned

Default QoS

The constant `DATAREADER_QOS_DEFAULT` can be used as parameter `qos` to create a `DataReader` with the default `DataReaderQos` as set in the `Subscriber`. The effect of using `DATAREADER_QOS_DEFAULT` is the same as calling the operation `get_default_datareader_qos` and using the resulting `DataReaderQos` to create the `DataReader`.

The special `DATAREADER_QOS_USE_TOPIC_QOS` can be used to create a `DataReader` with a combination of the default `DataReaderQos` and the `TopicQos`. The effect of using `DATAREADER_QOS_USE_TOPIC_QOS` is the same as calling the operation `get_default_datareader_qos` and retrieving the `TopicQos` (by means of the operation `Topic.get_qos`) and then combining these two `QosPolicy` settings using the operation `copy_from_topic_qos`, whereby any common policy that is set on the `TopicQos` "overrides" the corresponding policy on the default `DataReaderQos`. The resulting `DataReaderQos` is then applied to create the `DataReader`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `DataReaderListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `DataReaderListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_CHANGED_STATUS`
- `SUBSCRIPTION_MATCHED_STATUS`.

⚠ Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the

`NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

In case a communication status is not activated in the `mask` of the `DataReaderListener`, the `SubscriberListener` of the containing `Subscriber` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `SubscriberListener` of the containing `Subscriber` and a `DataReader` specific behaviour when needed. In case the communication status is not activated in the `mask` of the `SubscriberListener` as well, the communication status will be propagated to the `DomainParticipantListener` of the containing `DomainParticipant`. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

3.5.1.4 delete_contained_entities

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
   delete_contained_entities
      (void);
```

Description

This operation deletes all the `DataReader` objects that were created by means of the `create_datareader` operation on the `Subscriber`.

Parameters

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation deletes all the `DataReader` objects that were created by means of the `create_datareader` operation on the `Subscriber`. In other words, it deletes all contained `DataReader` objects. Prior to deleting each `DataReader`, this operation recursively calls the corresponding `delete_contained_entities` operation on each `DataReader`. In other words, all `DataReader` objects in the `Subscriber` are deleted, including the `QueryCondition` and `ReadCondition` objects contained by the `DataReader`.

⚠ **NOTE***:* The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained `DataReader` cannot be deleted because the application has called a `read` or `take` operation and has not called the corresponding `return_loan` operation to return the loaned samples. In such cases, the operation does not roll back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

- *RETCODE_OK* - the contained `Entity` objects are deleted and the application may delete the `Subscriber`;
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - one or more of the contained entities are in a state where they cannot be deleted.

3.5.1.5 **delete_datareader**

Scope

    DDS.Subscriber

Synopsis

    import DDS.*;

```
public int
   delete_datareader
      (DataReader a_datareader);
```

Description

This operation deletes a `DataReader` that belongs to the `Subscriber`.

Parameters

in DataReader a_datareader - a reference to the `DataReader`, which is to be deleted.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes a `DataReader` that belongs to the `Subscriber`. When the operation is called on a different `Subscriber`, as used when the `DataReader` was created, the operation has no effect and returns `RETCODE_PRECONDITION_NOT_MET`. The deletion of the `DataReader` is not allowed if there are any `ReadCondition` or `QueryCondition` objects that are attached to the `DataReader`, or when the `DataReader` still contains unreturned loans. In those cases the operation also returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `DataReader` is deleted

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter `a_datareader` is not a valid `DataReader`

- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `Subscriber` as used when the `DataReader` was created, the `DataReader` contains one or more `ReadCondition` or `QueryCondition` objects or the `DataReader` still contains unreturned loans.

3.5.1.6 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
    enable
        (void);
```

3.5.1.7 end_access

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
    end_access
        (void);
```

Description

This operation indicates that the application will stop accessing a coherent and/or ordered set of modifications that spans multiple DataReaders attached to this `Subscriber`. This access must have been started by a matching call to `begin_access`.

Parameters

<none>

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_ PRECONDITION_NOT_MET`.

Detailed Description

Indicates that the application has finished accessing the data samples in DataReader objects managed by the Subscriber. This operation must be used to 'close' a corresponding `begin_access`. The operation will effectively unlock all of the Subscriber's DataReader objects for incoming modifications, so it is important to invoke it as quickly as possible to avoid an ever increasing backlog of modifications. After calling `end_access` the application should no longer access any of the Data or SampleInfo elements returned from the sample-accessing operations.

PRISMTECH

This call must close a previous call to `begin_access` otherwise the operation will return the error `PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - access to coherent/ordered data has successfully started.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.

- *RETCODE_PRECONDITION_NOT_MET* - no matching call to `begin_access` has been detected.

3.5.1.8 get_datareaders

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
   get_datareaders
      (DataReaderSeqHolder readers,
        int sample_states,
        int view_states,
        int instance_states);
```

Description

This operation allows the application to access the DataReader objects that contain samples with the specified `sample_states`, `view_states`, and `instance_states`.

Parameters

inout DataReaderSeqHolder readers - a sequence which is used to pass the list of all DataReaders that contain samples of the specified `sample_states`, `view_states`, and `instance_states`.

in int sample_states - a mask, which selects only those readers that have samples with the desired sample states.

in int view_states - a mask, which selects only those readers that have samples with the desired view states.

in int instance_states - a mask, which selects only those readers that have samples with the desired instance states.

Return Value

int - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED, RETCODE_ OUT_OF_RESOURCES, RETCODE_NOT_ENABLED or RETCODE_ PRECONDITION_NOT_MET.

Detailed Description

This operation allows the application to access the DataReader objects that contain samples with the specified sample_states, view_states, and instance_states.

If the PresentationQosPolicy of the Subscriber to which the DataReader belongs has the access_scope set to 'GROUP', this operation should only be invoked inside a begin_access/end_access block. Otherwise it will return the error RETCODE_PRECONDITION_NOT_MET.

Depending on the setting of the PresentationQoSPolicy (see Section *3.1.3.14* on page 68), the returned collection of DataReader objects may be:

• a 'set' containing each DataReader at most once in no specified order,

• a 'list' containing each DataReader one or more times in a specific order.

This difference is due to the fact that, in the second situation it is required to access samples belonging to different DataReader objects in a particular order. In this case, the application should process each DataReader in the same order it appears in the 'list' and read or take exactly one sample from each DataReader. The patterns that an application should use to access data is fully described in Section 3.1.3.14, *PresentationQosPolicy*, on page 68.

It is allowed to pre-allocate the DataReader sequence prior to invoking this function:

• if the PresentationQosPolicy has access_scope set to 'GROUP' and ordered_access set to TRUE, the ReaderList will never contain more than the pre-allocated number of elements. Pre-allocating '*n*' Reader elements this way is convenient in scenario's where the application only intends to access the first '*n*' ordered samples. Otherwise the middleware would waste precious CPU cycles composing a list that will only partially be accessed.

• In all other cases, the ReaderList will re-allocated when it is not big enough to accommodate a list describing all DataReaders that have matching samples.

Return Code

When the operation returns:

• *RETCODE_OK* - a list of DataReaders matching your description has successfully been composed.

• *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the Subscriber is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is not invoked inside a begin_access/end_access block as required by its QosPolicy settings.

3.5.1.9 get_default_datareader_qos

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public void
   get_default_datareader_qos
      (DataReaderQosHolder qos);
```

Description

This operation gets the default QosPolicy settings of the DataReader.

Parameters

inout DataReaderQosHolder qos - a reference to the destination DataReaderQosHolder object in which the default DataReaderQos for the Subscriber is written.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation gets the default QosPolicy settings of the DataReader (that is the DataReaderQos) which is used for newly created DataReader objects, in case the constant DATAREADER_QOS_DEFAULT is used. The default DataReaderQos is only used when the constant is supplied as parameter qos to specify the DataReaderQos in the create_datareader operation. The application must provide the DataReaderQos object in which the QosPolicy settings can be stored and pass the qos reference to the operation. The operation writes the default QosPolicy settings to the object referenced to by qos. Any settings in the object are overwritten.

The values retrieved by this operation match the values specified on the last successful call to `set_default_datareader_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` setting as defined in Table 3, *QosPolicy Default Attributes*, on page 41.

Return Code

When the operation returns:

- *RETCODE_OK* - the default `DataReader` `QosPolicy` settings of this `Subscriber` have successfully been copied into the specified `DataReaderQosHolder` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.10 get_listener

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public SubscriberListener
   get_listener
        (void);
```

Description

This operation allows access to a `SubscriberListener`.

Parameters

<none>

Return Value

SubscriberListener - result is a reference to the `SubscriberListener` attached to the `Subscriber`.

Detailed Description

This operation allows access to a `SubscriberListener` attached to the `Subscriber`. When no `SubscriberListener` was attached to the `Subscriber`, the `null` reference is returned.

3.5.1.11 **get_participant**

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public DomainParticipant
   get_participant
       (void);
```

Description

This operation returns the `DomainParticipant` associated with the `Subscriber` or the `null` pointer.

Parameters

<none>

Return Value

DomainParticipant - a reference to the `DomainParticipant` associated with the `Subscriber` or the `null` pointer.

Detailed Description

This operation returns the `DomainParticipant` associated with the `Subscriber`. Note that there is exactly one `DomainParticipant` associated with each `Subscriber`. When the `Subscriber` was already deleted (there is no associated `DomainParticipant` any more), the `null` pointer is returned.

3.5.1.12 **get_qos**

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
   get_qos
       (SubscriberQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a `Subscriber`.

Parameters

inout SubscriberQosHolder qos - the destination `SubscriberQosHolder`
object in which the `QosPolicy` settings will be copied.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or
`RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `Subscriber`
on which this operation is used. This `SubscriberQos` is stored at the location
referenced by the `qos` parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this `Subscriber`
  has successfully been copied into the specified `SubscriberQosHolder`
  parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.5.1.13 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the interface
`Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
      (void);
```

3.5.1.14 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the class `Entity`
for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
```

```
get_statuscondition
   (void);
```

3.5.1.15 lookup_datareader

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public DataReader
   lookup_datareader
         (String topic_name);
```

Description

This operation returns a previously created `DataReader` belonging to the `Subscriber` which is attached to a `Topic` with the matching `topic_name`.

Parameters

in String topic_name - the name of the `Topic`, which is attached to the `DataReader` to look for.

Return Value

DataReader - Return value is a reference to the `DataReader` found. When no such `DataReader` is found, the `null` reference is returned.

Detailed Description

This operation returns a previously created `DataReader` belonging to the `Subscriber` which is attached to a `Topic` with the matching `topic_name`. When multiple `DataReader` objects (which satisfy the same condition) exist, this operation will return one of them. It is not specified which one.

This operation may be used on the built-in `Subscriber`, which returns the built-in `DataReader` objects for the built-in `Topics`.

3.5.1.16 notify_datareaders

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
   notify_datareaders
```

```
                (void);
```

Description

This operation invokes the `on_data_available` operation on `DataReaderListener` objects which are attached to contained `DataReader` entities and which have new, available data.

Parameters

<none>

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or
    `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation invokes the `on_data_available` operation for `DataReaderListener` objects that are attached to contained `DataReader` entities which have received information that has not yet been processed by those `DataReaders`.

The `notify_datareaders` operation ignores the bit mask value of the individual `DataReaderListener` objects, even when the `DATA_AVAILABLE_STATUS` bit has not been set on a `DataReader` that has new data available. The `on_data_available` operation will still be invoked, when the `DATA_AVAILABLE_STATUS` bit has not been set, but will not propagate to the `DomainParticipantListener`.

When the `DataReader` has attached a `NULL` listener, the event will be consumed and will not propagate to the `DomainParticipantListener`. (Remember that a `NULL` listener is regarded as a listener that handles all its events as a NOOP).

Return Code

When the operation returns:

- *RETCODE_OK* - all appropriate listeners have been invoked.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.17 **set_default_datareader_qos**

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
    set_default_datareader_qos
        (DataReaderQos qos);
```

Description

This operation sets the default `DataReaderQos` of the `DataReader`.

Parameters

in DataReaderQos qos - the `DataReaderQos` object, which contains the new default `QosPolicy` settings for the newly created `DataReaders`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_INCONSISTENT_POLICY`.

Detailed Description

This operation sets the default `DataReaderQos` of the `DataReader` (that is the struct with the `QosPolicy` settings). This `QosPolicy` is used for newly created `DataReader` objects in case the constant `DATAREADER_QOS_DEFAULT` is used as parameter `qos` to specify the `DataReaderQos` in the `create_datareader` operation. This operation checks if the `DataReaderQos` is self consistent. If it is not, the operation has no effect and returns `RETCODE_INCONSISTENT_POLICY`.

The values set by this operation are returned by `get_default_datareader_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `DataReaderQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid `DataReaderQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value or an enum value that is outside its legal boundaries.

- *RETCODE_UNSUPPORTED* - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_INCONSISTENT_POLICY* - the parameter `qos` contains conflicting `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.5.1.18 set_listener

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
  set_listener
     (SubscriberListener a_listener,
        int mask);
```

Description

This operation attaches a `SubscriberListener` to the `Subscriber`.

Parameters

in SubscriberListener a_listener - a reference to the `SubscriberListener` instance, which will be attached to the `Subscriber`.

in int mask - a bit mask in which each bit enables the invocation of the `SubscriberListener` for a certain status.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_ OF_RESOURCES`.

Detailed Description

This operation attaches a `SubscriberListener` to the `Subscriber`. Only one `SubscriberListener` can be attached to each `Subscriber`. If a `SubscriberListener` was already attached, the operation will replace it with the new one. When `a_listener` is the `null` reference, it represents a listener that is treated as a NOOP[1] for all statuses activated in the bitmask.

PRISMTECH

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `SubscriberListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset. An exception to this rule is the `null` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `SubscriberListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS`   *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`   *(propagated)*
- `SAMPLE_LOST_STATUS`   *(propagated)*
- `SAMPLE_REJECTED_STATUS`   *(propagated)*
- `DATA_AVAILABLE_STATUS`   *(propagated)*
- `LIVELINESS_CHANGED_STATUS`   *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS`   *(propagated)*.
- `DATA_ON_READERS_STATUS`.

Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the `PublisherListener`.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataReaderListener` of a contained `DataReader`, the `DataReaderListener`

---

1. Short for **No-Op**eration, an instruction that peforms nothing at all.

on that contained `DataReader` is invoked instead of the `SubscriberListener`. This means that a status change on a contained `DataReader` only invokes the `SubscriberListener` if the contained `DataReader` itself does not handle the trigger event generated by the status change.

In case a communication status is not activated in the `mask` of the `SubscriberListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Subscriber` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are "Read Communication Statuses" and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a DataReader, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipant Listener` for the `DATA_AVAILABLE_STATUS` (in that order).

Return Code

When the operation returns:

- *RETCODE_OK* - the `SubscriberListener` is attached
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_UNSUPPORTED* - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.19 set_qos

Scope

```
DDS.Subscriber
```

Synopsis

```
import DDS.*;
public int
    set_qos
        (SubscriberQos qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `Subscriber`.

Parameters

in SubscriberQos qos - the new set of `QosPolicy` settings for the `Subscriber`.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`,
    `RETCODE_UNSUPPORTED`,   `RETCODE_ALREADY_DELETED`,
    `RETCODE_OUT_OF_RESOURCES`, `RETCODE_IMMUTABLE_POLICY` or
    `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `Subscriber`. The parameter `qos` contains the object with the `QosPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QosPolicy` setting for an enabled `Subscriber`, which can only be set before the `Subscriber` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the presently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` settings (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

The set of `QosPolicy` settings specified by the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned `RETCODE_OK`). If one or more of the partitions in the `QoS` structure have insufficient access rights configured then the `set_qos` function will fail with a `RETCODE_PRECONDITION_NOT_MET` error code.

Return Code

When the operation returns:

- *RETCODE_OK* - the new `SubscriberQos` is set

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid SubscriberQos.
- *RETCODE_UNSUPPORTED* - one or more of the selected QosPolicy values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter qos contains an immutable QosPolicy setting with a different value than set during enabling of the Subscriber.
- *RETCODE_PRECONDITION_NOT_MET* - returned when insufficient access rights exist for the partition(s) listed in the QoS structure.

3.5.2 Subscription Type Specific Classes

"Subscription type specific classes" contains the generic class and the generated data type specific classes. For each data type, a data type specific class <type>DataReader is generated (based on IDL) by calling the pre-processor. In case of data type Foo (this also applies to other types); "Subscription type specific classes" contains the following classes:

This paragraph describes the generic DataReader class and the derived application type specific <type>DataReader classes which together implement the application subscription interface. For each application type, used as Topic data type, the pre-processor generates a <type>DataReader class from an IDL type description. The FooDataReader class that would be generated by the pre-processor for a fictional type Foo describes the <type>DataReader class.

3.5.2.1 Interface DataReader

A DataReader allows the application:

- to declare data it wishes to receive (i.e., make a subscription);
- to access data received by the associated Subscriber.
- A DataReader refers to exactly one TopicDescription (either a Topic, a ContentFilteredTopic or a MultiTopic) that identifies the samples to be read. The DataReader may give access to several instances of the data type, which are distinguished from each other by their key.

DataReader is an interface. It is specialized for each particular application data type. For a fictional application data type "Foo" (defined in the module SPACE) the specialized class would be SPACE.FooDataReader.

The interface description is as follows:

```
public interface DataReader
{
```

PRISMTECH

```
//
// extends interface class Entity
//
// public StatusCondition
//    get_statuscondition
//        (void);
// public int
//    get_status_changes
//        (void);
// public int
//    enable
//        (void);
//
// operations (implemented in the data type specific DataReader)
//
// public int
//    read
//        (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//    take
//        (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//    read_w_condition
//        (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          ReadCondition a_condition);
// public int
//    take_w_condition
//        (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          ReadCondition a_condition);
// public int
//    read_next_sample
//        (<type>Holder data_value,
//          SampleInfoHolder sample_info);
// public int
//    take_next_sample
//        (<type>Holder data_value,
```

```
//            SampleInfoHolder sample_info);
// public int
//     read_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     take_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     read_next_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     take_next_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     read_next_instance_w_condition
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          ReadCondition a_condition);
// public int
//     take_next_instance_w_condition
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
```

```
//           ReadCondition a_condition);
// public int
//     return_loan
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq);
// public int
//     get_key_value
//         (<type>Holder key_holder,
//          long handle);
// public long
//     lookup_instance
//         (<type> instance_data);
//
// implemented API operations
//
    public ReadCondition
       create_readcondition
          (int sample_states,
           int view_states,
           int instance_states);

    public QueryCondition
       create_querycondition
          (int sample_states,
           int view_states,
           int instance_states,
           String query_expression,
           String[] query_parameters);

    public int
       delete_readcondition
          (ReadCondition a_condition);

    public int
       delete_contained_entities
          (void);

    public int
       set_qos
          (DataReaderQos qos);

    public int
       get_qos
          (DataReaderQosHolder qos);

    public int
       set_listener
          (DataReaderListener a_listener,
            int mask);
```

```
public DataReaderListener
   get_listener
      (void);

public TopicDescription
   get_topicdescription
      (void);

public Subscriber
   get_subscriber
      (void);

public int
   get_sample_rejected_status
      (SampleRejectedStatusHolder status);

public int
   get_liveliness_changed_status
      (LivelinessChangedStatusHolder status);

public int
   get_requested_deadline_missed_status
      (RequestedDeadlineMissedStatusHolder status);

public int
   get_requested_incompatible_qos_status
      (RequestedIncompatibleQosStatusHolder status);

public int
   get_subscription_matched_status
      (SubscriptionMatchedStatusHolder status);

public int
   get_sample_lost_status
      (SampleLostStatusHolder status);

public int
   wait_for_historical_data
      (Duration_t max_wait);

public int
   get_matched_publications
      (InstanceHandleSeqHolder publication_handles);

public int
   get_matched_publication_data
      (PublicationBuiltinTopicDataHolder publication_data,
      long publication_handle);

DataReaderView
```

PRISMTECH

```
        create_view
          (DataReaderViewQos qos);

    int
        delete_view
          (DataReaderView a_view);

    int
        get_default_datareaderview_qos
          (DataReaderViewQosHolder qos);

    int
        set_default_datareaderview_qos
          (DataReaderViewQos qos);
};
```

The following paragraphs describe the usage of all `DataReader` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited. The abstract operations are listed but not fully described because they are not implemented in this specific class. The full description of these operations is located in the subclasses that contain the data type specific implementation of these operations.

3.5.2.2 create_querycondition

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public QueryCondition
    create_querycondition
        (int sample_states,
         int view_states,
         int instance_states,
         String query_expression,
         String[] query_parameters);
```

Description

This operation creates a new `QueryCondition` for the `DataReader`.

Parameters

in int sample_states - a mask, which selects only those samples with the desired sample states.

in *int view_states* - a mask, which selects only those samples with the desired view states.

in *int instance_states* - a mask, which selects only those samples with the desired instance states.

in *String query_expression* - the query string, which must be a subset of the SQL query language.

in *String[] query_parameters* - a sequence of strings which are the parameter values used in the SQL query string (i.e., the "%n" tokens in the expression). The number of values in query_parameters must be equal or greater than the highest referenced %n token in the query_expression (e.g. if %1 and %8 are used as parameter in the query_expression, the query_parameters should at least contain n+1 = 9 values).

Return Value

QueryCondition - Result value is a reference to the QueryCondition. When the operation fails, the null reference is returned.

Detailed Description

This operation creates a new QueryCondition for the DataReader. The returned QueryCondition is attached (and belongs) to the DataReader. When the operation fails, the null reference is returned. To delete the QueryCondition the operation delete_readcondition or delete_contained_entities must be used.

State Masks

The result of the QueryCondition also depends on the selection of samples determined by three masks:

- sample_states is the mask, which selects only those samples with the desired sample states READ_SAMPLE_STATE, NOT_READ_SAMPLE_STATE or both

- view_states is the mask, which selects only those samples with the desired view states NEW_VIEW_STATE, NOT_NEW_VIEW_STATE or both

- instance_states is the mask, which selects only those samples with the desired instance states ALIVE_INSTANCE_STATE, NOT_ALIVE_DISPOSED_INSTANCE_STATE, NOT_ALIVE_NO_WRITERS_INSTANCE_STATE or a combination of these.

SQL Expression

The SQL query string is set by query_expression which must be a subset of the SQL query language. In this query expression, parameters may be used, which must be set in the sequence of strings defined by the parameter query_parameters. A

parameter is a string which can define an integer, float, string or enumeration. The number of values in `query_parameters` must be equal or greater than the highest referenced %n token in the `query_expression` (e.g. if `%1` and `%8` are used as parameter in the `query_expression`, the `query_parameters` should at least contain `n+1 = 9 values`).

3.5.2.3 **create_readcondition**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public ReadCondition
    create_readcondition
        (int sample_states,
         int view_states,
         int instance_states);
```

Description

This operation creates a new `ReadCondition` for the `DataReader`.

Parameters

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

in int instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReadCondition - Result value is a reference to the `ReadCondition`. When the operation fails, the `null` reference is returned.

Detailed Description

This operation creates a new `ReadCondition` for the `DataReader`. The returned `ReadCondition` is attached (and belongs) to the `DataReader`. When the operation fails, the `null` reference is returned. To delete the `ReadCondition` the operation `delete_readcondition` or `delete_contained_entities` must be used.

State Masks

The result of the `ReadCondition` depends on the selection of samples determined by three masks:

- `sample_states` is the mask, which selects only those samples with the desired sample states `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both

- `view_states` is the mask, which selects only those samples with the desired view states `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE` or both

- `instance_states` is the mask, which selects only those samples with the desired instance states `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these.

3.5.2.4 **create_view**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
DataReaderView
  create_view
    (DataReaderViewQos qos);
```

Description

This operation creates a DataReaderView with the desired QosPolicy settings.

Parameters

in DataReaderViewQos - the QosPolicy settings for the DataReaderView.

Return Value

DataReaderView - The newly-created DataReaderView. In case of error, `null` is returned.

Detailed Description

This operation creates a DataReaderView with the desired QosPolicy settings. In case the QosPolicy is invalid, `null` is returned. The convenience macro `DATAREADERVIEW_QOS_DEFAULT` can be used as parameter qos, to create a DataReaderView with the default DataReaderViewQos as set in the DataReader.

Application Data Type

The DataReaderView returned by this operation is an object of a derived class, specific to the data type associated with the Topic. For each application-defined data type `<type>` there is a class `<type>DataReaderView` generated by calling the pre-processor. This data type specific class extends DataReaderView and contains the operations to read and take data of data type `<type>`.

The typed operations of a DataReaderView exactly mimic those of the DataReader from which it is created.

3.5.2.5 delete_contained_entities

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
    delete_contained_entities
        (void);
```

Description

This operation deletes all the `Entity` objects that were created by means of one of the "`create_`" operations on the `DataReader`.

Parameters

<none>

Return Value

`int` - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.
```

Detailed Description

This operation deletes all the `Entity` objects that were created by means of one of the "`create_`" operations on the `DataReader`. In other words, it deletes all `QueryCondition` and `ReadCondition` objects contained by the `DataReader`.

NOTE*:* The operation will return `PRECONDITION_NOT_MET` if the any of the contained entities is in a state where it cannot be deleted. In such cases, the operation does not roll back any entity deletions performed prior to the detection of the problem.

When the operation returns:

- *RETCODE_OK* - the contained `Entity` objects are deleted and the application may delete the `DataReader`
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - one or more of the contained entities are in a state where they cannot be deleted.

3.5.2.6 delete_readcondition

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   delete_readcondition
      (ReadCondition a_condition);
```

Description

This operation deletes a `ReadCondition` or `QueryCondition` which is attached to the `DataReader`.

Parameters

in ReadCondition a_condition - a reference to the `ReadCondition` or `QueryCondition` which is to be deleted.

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
   RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or
   RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation deletes a `ReadCondition` or `QueryCondition` which is attached to the `DataReader`. Since a `QueryCondition` is a specialized `ReadCondition`, the operation can also be used to delete a `QueryCondition`. A `ReadCondition` or `QueryCondition` cannot be deleted when it is not attached to this `DataReader`.

When the operation is called on a `ReadCondition` or `QueryCondition` which was not attached to this `DataReader`, the operation returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `ReadCondition` or `QueryCondition` is deleted
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter a_condition is not a valid `ReadCondition`
- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DataReader`, as used when the `ReadCondition` or `QueryCondition` was created.

3.5.2.7 delete_view

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
int
   delete_view
      (DataReaderView a_view);
```

Description

This operation deletes a DataReaderView that belongs to the DataReader.

Parameters

in DataReaderView a_view - The DataReaderView which is to be deleted.

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,
RETCODE_PRECONDITION_NOT_MET.
```

Detailed Description

This operation deletes the DataReaderView from the DataReader.

Return Code

When the operation returns:

- *RETCODE_OK* - the DataReaderView is deleted.
- *RETCODE_ERROR* - an internal error occurred.
- *RETCODE_BAD_PARAMETER* - the DataReaderView_ptr parameter is invalid.
- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the data distribution service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the DataReaderView is not associated with this DataReader, or the DataReaderView still contains one or more ReadCondition or QueryCondition objects or an unreturned loan.

3.5.2.8 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   enable
      (void);
```

3.5.2.9 get_default_datareaderview_qos

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
int
   get_default_datareaderview_qos
      (DataReaderViewQosHolder qos);
```

Description

This operation gets the default QosPolicy settings of the DataReaderView.

Parameters

in DataReaderViewQosHolder qos - a DataReaderViewQosHolder that holds the DataReaderViewQos in which the default QosPolicy settings will be stored.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation gets the default QosPolicy settings of the DataReaderView, which are used for newly-created DataReaderView objects in case the constant DATAREADERVIEW_QOS_DEFAULT is used.

The values retrieved by this call match the values specified on the last successful call to set_default_datareaderview_qos, or, if this call was never made, the default values as specified in Table 3, *QosPolicy Default Attributes*, on page 41.

Return Code

When the operation returns:

- *RETCODE_OK* - the default DataReaderView QosPolicy settings of this DataReader have successfully been copied into the provided DataReaderViewQos parameter.

- *RETCODE_ERROR* - an internal error has occurred

- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the data distribution service ran out of resources to complete this operation

3.5.2.10 get_key_value (abstract)

This operation is defined as a generic operation, which is implemented by the <type>DataReader class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type Foo derived FooDataReader class.

Synopsis

```
import DDS.*;
public int
    get_key_value
        (<type>Holder key_holder,
        long handle);
```

3.5.2.11 get_listener

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public DataReaderListener
   get_listener
       (void);
```

Description

This operation allows access to a `DataReaderListener`.

Parameters

<none>

Return Value

DataReaderListener - result is a reference to the `DataReaderListener` attached to the `DataReader`.

Detailed Description

This operation allows access to a `DataReaderListener` attached to the `DataReader`. When no `DataReaderListener` was attached to the `DataReader`, the `null` reference is returned.

3.5.2.12 get_liveliness_changed_status

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_liveliness_changed_status
       (LivelinessChangedStatusHolder status);
```

Description

This operation obtains the `LivelinessChangedStatus` object of the `DataReader`.

Parameters

inout LivelinessChangedStatusHolder status - the contents of the
LivelinessChangedStatus object of the DataReader will be copied into
the LivelinessChangedStatusHolder specified by status.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or
RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation obtains the LivelinessChangedStatus object of the
DataReader. This object contains the information whether the liveliness of one or
more DataWriter objects that were writing instances read by the DataReader has
changed. In other words, some DataWriter have become "alive" or "not alive".

The LivelinessChangedStatus can also be monitored using a
DataReaderListener or by using the associated StatusCondition.

Return Code

When the operation returns:

- *RETCODE_OK* - the current LivelinessChangedStatus of this DataReader
  has successfully been copied into the specified status parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of
  resources to complete this operation.

3.5.2.13 **get_matched_publication_data**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
  get_matched_publication_data
     (PublicationBuiltinTopicDataHolder publication_data,
      long publication_handle);
```

Description

This operation retrieves information on the specified publication that is currently "associated" with the `DataReader`.

Parameters

inout PublicationBuiltinTopicDataHolder publication_data - a Holder for the sample in which the information about the specified publication is to be stored.

in long publication_handle - a handle to the publication whose information needs to be retrieved.

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_ OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation retrieves information on the specified publication that is currently "associated" with the `DataReader`. That is, a publication with a matching Topic and compatible QoS that the application has not indicated should be "ignored" by means of the `ignore_publication` operation on the `DomainParticipant`.

The `publication_handle` must correspond to a publication currently associated with the `DataReader`, otherwise the operation will fail and return `RETCODE_BAD_PARAMETER`. The operation `get_matched_publications` can be used to find the publications that are currently matched with the `DataReader`.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the `publication_data`. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

Return Code

When the operation returns:

- *RETCODE_OK* - the information on the specified publication has successfully been retrieved.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" publications.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `DataReader` is not enabled.

3.5.2.14 get_matched_publications

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_matched_publications
      (InstanceHandleSeqHolder publication_handles);
```

Description

This operation retrieves the list of publications currently "associated" with the `DataReader`.

Parameters

inout InstanceHandleSeqHolder publication_handles - a Holder for an array which is used to pass the list of all associated publications.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK,   RETCODE_ERROR,   RETCODE_UNSUPPORTED,`
   `RETCODE_ALREADY_DELETED, RETCODE_OUT_ OF_RESOURCES` or
   `RETCODE_NOT_ENABLED.`

Detailed Description

This operation retrieves the list of publications currently "associated" with the `DataReader`. That is, subscriptions that have a matching Topic and compatible QoS that the application has not indicated should be "ignored" by means of the `ignore_publication` operation on the `DomainParticipant`.

The array inside the `publication_handles` Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the `get_matched_publications` operation. If the pre-allocated array is not big enough to hold the number of associated publications, the array will automatically be (re-)allocated to fit the required size.

The handles returned in the `publication_handles` array are the ones that are used by the DDS implementation to locally identify the corresponding matched publication entities. You can access more detailed information about a particular publication by passing its `publication_handle` to either the `get_matched_publication_data` operation or to the `read_instance` operation on the built-in reader for the "DCPSPublication" topic.

Be aware that since an instance handle is an opaque datatype, it does not necessarily mean that the handles obtained from the `get_matched_publications` operation have the same value as the ones that appear in the `instance_handle` field of the `SampleInfo` when retrieving the publication info through corresponding "DCPSPublications" built-in reader. You can't just compare two handles to determine whether they represent the same publication. If you want to know whether two handles actually do represent the same publication, use both handles to retrieve their corresponding `PublicationBuiltinTopicData` samples and then compare the `key` field of both samples.

The operation may fail if the infrastructure does not locally maintain the connectivity information. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

Return Code

When the operation returns:

- *RETCODE_OK* - the list of associated publications has successfully been obtained.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" publications.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `DataReader` is not enabled.

3.5.2.15 **get_property**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
  get_property
    (PropertyHolder a_property);
```

Description

This function queries the value of a property set on a `DataReader`.

Parameters

inout PropertyHolder a_property - on entry, the name of the property in the holder determines which property to query the value of; on successful return, the value of that property is set to the current value of that property in the DataReader.

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES, RETCODE_UNSUPPORTED
```

Detailed Description

This operation looks up the property specified by `a_property.property.name` in the DataReader, setting `a_property.property.value` to the current value of the property. If the property has not been set using `set_property`, the default value is returned.

Return Code

When the operation returns:

- *RETCODE_OK* - a_property.property.value has been set to the current value of the property.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_UNSUPPORTED* - a_property.property.name specifies an undefined property or the operation is not supported in this version.

3.5.2.16 **get_qos**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_qos
      (DataReaderQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a `DataReader`.

Parameters

inout DataReaderQosHolder qos - a reference to the destination `DataReaderQosHolder` object in which the `QosPolicy` settings will be copied.

Return Value

int - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `DataReader` on which this operation is used. This `DataReaderQos` is stored at the location referenced by the `qos` parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of `QoSPolicy` values applied to this `DataReader` has successfully been copied into the specified `DataReaderQosHolder` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.17 **get_requested_deadline_missed_status**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_requested_deadline_missed_status
      (RequestedDeadlineMissedStatusHolder status);
```

Description

This operation obtains the `RequestedDeadlineMissedStatus` object of the `DataReader`.

Parameters

inout RequestedDeadlineMissedStatusHolder status - the contents of the `RequestedDeadlineMissedStatus` object of the `DataReader` will be copied into the `RequestedDeadlineMissedStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `RequestedDeadlineMissedStatus` object of the `DataReader`. This object contains the information whether the deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected for a specific instance.

The `RequestedDeadlineMissedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `RequestedDeadlineMissedStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.18 get_requested_incompatible_qos_status

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_requested_incompatible_qos_status
      (RequestedIncompatibleQosStatusHolder status);
```

Description

This operation obtains the `RequestedIncompatibleQosStatus` object of the `DataReader`.

Parameters

inout RequestedIncompatibleQosStatusHolder status - the contents of the `RequestedIncompatibleQosStatus` object of the `DataReader` will be copied into the `RequestedIncompatibleQosStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `RequestedIncompatibleQosStatus` object of the `DataReader`. This object contains the information whether a `QosPolicy` setting was incompatible with the offered `QosPolicy` setting.

The Request/Offering mechanism is applicable between the `DataWriter` and the `DataReader`. If the `QosPolicy` settings between `DataWriter` and `DataReader` are inconsistent, no communication between them is established. In addition the `DataWriter` will be informed via a `REQUESTED_INCOMPATIBLE_QOS` status change and the `DataReader` will be informed via an `OFFERED_INCOMPATIBLE_QOS` status change.

The `RequestedIncompatibleQosStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `RequestedIncompatibleQosStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.19 get_sample_lost_status

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
  get_sample_lost_status
      (SampleLostStatusHolder status);
```

Description

This operation obtains the `SampleLostStatus` object of the `DataReader`.

Parameters

inout SampleLostStatusHolder status - the contents of the `SampleLostStatus` object of the `DataReader` will be copied into the `SampleLostStatusHolder` specified by `status`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `SampleLostStatus` object of the `DataReader`. This object contains information whether samples have been lost. This only applies when the `ReliabilityQosPolicy` is set to `RELIABLE`. If the `ReliabilityQosPolicy` is set to `BEST_EFFORT` the Data Distribution Service will not report the loss of samples.

The `SampleLostStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current SampleLostStatus of this DataReader has successfully been copied into the specified status parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.20 get_sample_rejected_status

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_sample_rejected_status
      (SampleRejectedStatusHolder status);
```

Detailed Description

This operation obtains the SampleRejectedStatus object of the DataReader.

Parameters

inout SampleRejectedStatusHolder status - the contents of the SampleRejectedStatus object of the DataReader will be copied into the SampleRejectedStatusHolder specified by status.

Return Value

int - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation obtains the SampleRejectedStatus object of the DataReader. This object contains the information whether a received sample has been rejected. Samples may be rejected by the DataReader when it runs out of resource_limits to store incoming samples. Ususally this means that old samples need to be 'consumed' (for example by 'taking' them instead of 'reading' them) to make room for newly incoming samples.

The `SampleRejectedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `SampleRejectedStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.21 **get_status_changes (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
       (void);
```

3.5.2.22 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
       (void);
```

3.5.2.23 **get_subscriber**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public Subscriber
   get_subscriber
       (void);
```

Description

This operation returns the `Subscriber` to which the `DataReader` belongs.

Parameters

<none>

Return Value

`Subscriber` - Return value is a reference to the `Subscriber` object to which the `DataReader` belongs.

Detailed Description

This operation returns the `Subscriber` to which the `DataReader` belongs, thus the `Subscriber` that has created the `DataReader`. If the `DataReader` is already deleted, the `null` reference is returned.

3.5.2.24 get_subscription_matched_status

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   get_subscription_match_status
      (SubscriptionMatchedStatusHolder status);
```

Description

This operation obtains the `SubscriptionMatchedStatus` object of the `DataReader`.

Parameters

`inout SubscriptionMatchedStatusHolder status` - the contents of the `SubscriptionMatchedStatus` object of the `DataReader` will be copied into the `SubscriptionMatchedStatusHolder` specified by `status`.

Return Value

`int` - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED` or `RETCODE_ OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `SubscriptionMatchedStatus` object of the `DataReader`. This object contains the information whether a new match has been discovered for the current subscription, or whether an existing match has ceased to exist.

This means that the status represents that either a DataWriter object has been discovered by the `DataReader` with the same Topic and a compatible Qos, or that a previously discovered DataWriter has ceased to be matched to the current `DataReader`. A DataWriter may cease to match when it gets deleted, when it changes its Qos to a value that is incompatible with the current `DataReader` or when either the `DataReader` or the DataWriter has chosen to put its matching counterpart on its ignore-list using the `ignore_publication` or `ignore_subcription` operations on the `DomainParticipant`.

The operation may fail if the infrastructure does not hold the information necessary to fill in the `SubscriptionMatchedStatus`. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

The `SubscriptionMatchedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the current `SubscriptionMatchedStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_UNSUPPORTED* - OpenSplice is configured not to maintain the information about "associated" subscriptions.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.25 get_topicdescription

Scope

> DDS.DataReader

Synopsis

> import DDS.*;

```
public TopicDescription
   get_topicdescription
       (void);
```

Description

This operation returns the `TopicDescription` which is associated with the `DataReader`.

Parameters

<none>

Return Value

TopicDescription - Return value is a reference to the `TopicDescription` object which is associated with the `DataReader`.

Detailed Description

This operation returns the `TopicDescription` which is associated with the `DataReader`, thus the `TopicDescription` with which the `DataReader` is created. If the `DataReader` is already deleted, the `null` reference is returned.

3.5.2.26 **lookup_instance (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public long
   lookup_instance
       (<type> instance_data);
```

3.5.2.27 **read (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   read
       (<type>SeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        int sample_states,
```

```
                    int view_states,
                    int instance_states);
```

3.5.2.28 read_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be
used. For further explanation see the description for the fictional data type `Foo`
interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   read_instance
       (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

3.5.2.29 read_next_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be
used. For further explanation see the description for the fictional data type `Foo`
interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   read_next_instance
       (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

3.5.2.30 read_next_instance_w_condition (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be
used. For further explanation see the description for the fictional data type `Foo`
interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   read_next_instance_w_condition
```

```
(<type>SeqHolder data_values,
 SampleInfoSeqHolder info_seq
 int max_samples,
 long a_handle,
 ReadCondition a_condition);
```

3.5.2.31 read_next_sample (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   read_next_sample
      (<type>Holder data_value,
        SampleInfoHolder sample_info);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.32 read_w_condition (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   read_w_condition
      (<type>SeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        ReadCondition a_condition);
```

3.5.2.33 return_loan (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   return_loan
      (<type>SeqHolder data_values,
        SampleInfoSeqHolder info_seq);
```

3.5.2.34 **set_default_datareaderview_qos**

Synopsis

```
import DDS.*;
   set_default_datareaderview_qos
      (DataReaderViewQos qos);
```

Description

This operation sets the default `DataReaderViewQos` of the DataReader.

Parameters

in `DataReaderViewQos qos` - the `DataReaderViewQos` which contains the
default QosPolicy settings for newly-created DataReaderView objects.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`,
`RETCODE_OUT_OF_RESOURCES`.

Detailed Description

Return Code

When the operation returns:

- *RETCODE_OK* - the new default DataReaderViewQos is set.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the `DataReaderViewQos` parameter is invalid.

- *RETCODE_OUT_OF_RESOURCES* - the data distribution service ran out of
resources to complete this operation.

3.5.2.35 **set_listener**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
   set_listener
      (DataReaderListener a_listener,
        int mask);
```

Description

This operation attaches a `DataReaderListener` to the `DataReader`.

Parameters

in DataReaderListener a_listener - a reference to the `DataReaderListener` instance, which will be attached to the `DataReader`.

in int mask - a bit mask in which each bit enables the invocation of the `DataReaderListener` for a certain status.

Return Value

int - Possible return codes of the operation are:
   `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_ OF_RESOURCES`.

Detailed Description

This operation attaches a `DataReaderListener` to the `DataReader`. Only one `DataReaderListener` can be attached to each `DataReader`. If a `DataReaderListener` was already attached, the operation will replace it with the new one. When `a_listener` is the `null` reference, it represents a listener that is treated as a NOOP[1] for all statuses activated in the bitmask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever that communication status changes. For each communication status activated in the `mask`, the associated `DataReaderListener` operation is invoked and the communication status is reset to `false`, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset. An exception to this rule is the `null` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `DataReaderListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_CHANGED_STATUS`

---

1.   Short for **No-Op**eration, an instruction that peforms nothing at all.

PRISMTECH

- SUBSCRIPTION_MATCHED_STATUS.

⚠ Be aware that the SUBSCRIPTION_MATCHED_STATUS is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the NetworkingService/Discovery/enabled property in the Deployment Manual for more information about this subject.) In this case the operation will return RETCODE_UNSUPPORTED.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant STATUS_MASK_NONE can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant STATUS_MASK_ANY_V1_2 can be used to select all applicable statuses specified in the "Data Distribution Service for Real-time Systems Version 1.2" specification which are applicable to the PublisherListener.

Status Propagation

In case a communication status is not activated in the mask, the SubscriberListener of the DataReaderListener is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the SubscriberListener of the containing Subscriber and a DataReader specific behaviour when needed. In case the communication status is not activated in the mask of the SubscriberListener as well, the communication status will be propagated to the DomainParticipantListener of the containing DomainParticipant. In case the DomainParticipantListener is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

The statuses DATA_ON_READERS_STATUS and DATA_AVAILABLE_STATUS are "Read Communication Statuses" and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a get_<status_name>_status operation and they are mutually exclusive. When new information becomes available to a DataReader, the Data Distribution Service will first look in an attached and activated SubscriberListener or DomainParticipantListener (in that order) for the DATA_ON_READERS_STATUS. In case the DATA_ON_READERS_STATUS can not be handled, the Data Distribution Service will look in an attached and activated DataReaderListener, SubscriberListener or DomainParticipant Listener for the DATA_AVAILABLE_STATUS (in that order).

Return Code

When the operation returns:

- *RETCODE_OK* - the `DataReaderListener` is attached

- *RETCODE_ERROR* - an internal error has occurred

- *RETCODE_UNSUPPORTED* - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.

- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.36 set_property

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
  set_property
    (Property property);
```

Description

This function sets a property on a `DataReader`.

Parameters

inout Property a_property - specifies the property (in `a_property.name`) and its new value (in `a_property.value`).

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_BAD_PARAMETER, RETCODE_ERROR,
RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,
RETCODE_UNSUPPORTED
```

Detailed Description

This operation sets the property specified by `a_property.name` to the value given by `a_property.value`.

Currently, the following properties are defined:

parallelReadThreadCount

   By default, the demarshalling of data into Java objects by a single read or take operation happens only in the calling thread. The `parallelReadThreadCount` property can be used to control the number of

parallel threads to be used for this demarshalling. When reading multiple of samples takes a significant amount of time, increasing the number of threads on a multi-core machine can provide a significant benefit.

The value is interpreted as the number of parallel threads to use (*i.e.*, the value is a string representing a natural integer in decimal notation, so for example the string '`4`' will cause 4 threads to be used). The value '`0`' is allowed and selects the default behaviour.

If the call was successful, successive read/take operations on that datareader will use the specified number of threads for the demarshalling step of the respective operations until the value of this property is changed again.

CDRCopy

The default demarshalling technique constructs the Java object directly from the shared memory, but for some types, it is more efficient to marshal the value in shared memory into CDR representation, and then demarshal the CDR representation from Java. The value is interpreted as a boolean (*i.e.*, value must be either '`true`' or '`false`'), with '`false`' selecting the default technique and '`true`' selecting the alternative, CDR-based technique.

The CDR-based technique requires JacORB to be configured as the default ORB in the JVM, and furthermore requires that the JacORB IDL preprocessor has been used to generate a Helper class for the topic type of this DataReader and that this helper class can be found in the class path. The ORB initialisation and Helper class lookup is all done at run-time to avoid introducing a compile-time dependency, but this means that attempting to set the `CDRCopy` property to '`true`' can fail in complicated ways because of these dependencies.

When set to `true`, the CDR-based technique will be used for successive read/take operations on the data reader, until the property is set to `false` again.

ignoreLoansOnDeletion

By default, the DataReader keeps history about open loans.

'Open loans' is memory given to the user by a take or a read action on a DataReader which is not returned with a call to `return_loan`.

A DataReader can by default not be deleted if there are open loans; such an attempt will result in a `RETCODE_PRECONDITION_NOT_MET` returncode.

To ignore open loans when it is necessary to delete the DataReader, set the property `ignoreLoansOnDeletion` on the DataReader to `true`. If this property is set the DataReader will ignore its open loans in the case of a delete action.

The value is interpreted as a boolean (*i.e.*, it must be either '`true`' or '`false`').

> **false** (default): The DataReader will check for open loans in case of a delete action and will return RETCODE_PRECONDITION_NOT_MET if there are open loans.
>
> **true**: The DataReader will ignore open loans and can be deleted.

Return Code

When the operation returns:

- *RETCODE_OK* - the property has been set.
- *RETCODE_BAD_PARAMETER* - an invalid value has been specified.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_UNSUPPORTED* - a_property.name specifies an undefined property or the operation is not supported in this version.

3.5.2.37 set_qos

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
  set_qos
     (DataReaderQos qos);
```

Description

This operation replaces the existing set of QosPolicy settings for a DataReader.

Parameters

in DataReaderQos qos - the new set of QosPolicy settings for the DataReader.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
RETCODE_UNSUPPORTED, RETCODE_ALREADY_ DELETED,
RETCODE_OUT_OF_RESOURCES, RETCODE_IMMUTABLE_POLICY or
RETCODE_INCONSISTENT_POLICY.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `DataReader`. The parameter `qos` contains the `QosPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QosPolicy` setting for an enabled `DataReader`, which can only be set before the `DataReader` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the presently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` setting (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

The set of `QosPolicy` settings specified by the `qos` parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned `RETCODE_OK`).

Return Code

When the operation returns:

- *RETCODE_OK* - the new `DataReaderQos` is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `qos` is not a valid `DataReaderQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value.
- *RETCODE_UNSUPPORTED* - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter `qos` contains an immutable `QosPolicy` setting with a different value than set during enabling of the `DataReader`
- *RETCODE_INCONSISTENT_POLICY* - the parameter `qos` contains conflicting `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.5.2.38 take (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
    take
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
```

3.5.2.39 **take_instance (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
    take_instance
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

3.5.2.40 **take_next_instance (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
    take_next_instance
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

PRISMTECH

3.5.2.41 **take_next_instance_w_condition (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   take_next_instance_w_condition
      (<type>SeqHolder data_values,
       SampleInfoSeqHolder info_seq
       int max_samples,
       long a_handle,
       ReadCondition a_condition);
```

3.5.2.42 **take_next_sample (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   take_next_sample
      (<type>Holder data_value,
       SampleInfoHolder sample_info);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.43 **take_w_condition (abstract)**

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

Synopsis

```
import DDS.*;
public int
   take_w_condition
      (<type>SeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       ReadCondition a_condition);
```

3.5.2.44 **wait_for_historical_data**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
public int
    wait_for_historical_data
        (Duration_t max_wait);
```

Description

This operation will block the application thread until all "historical" data is received.

Parameters

in Duration_t max_wait - the maximum duration to block for the
wait_for_historical_data, after which the application thread is
unblocked. The special constant DURATION_INFINITE can be used when the
maximum waiting time does not need to be bounded.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED,
RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED or
RETCODE_TIMEOUT.

Detailed Description

This operation behaves differently for DataReader objects which have a
non-VOLATILE_DURABILITY_QOS DurabilityQosPolicy and for
DataReader objects which have a VOLATILE_DURABILITY_QOS
DurabilityQosPolicy.

As soon as an application enables a non-VOLATILE_DURABILITY_QOS
DataReader it will start receiving both "historical" data, i.e. the data that was
written prior to the time the DataReader joined the domain, as well as any new
data written by the DataWriter objects. 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 wait_for_historical_data operation.

As soon as an application enables a VOLATILE_DURABILITY_QOS DataReader it
will not start receiving "historical" data but only new data written by the
DataWriter objects. By calling wait_for_historical_data the DataReader
explicitly requests the Data Distribution Service to start receiving also the

PRISMTECH

"historical" data and to wait until either all "historical" data is received, or the duration specified by the `max_wait` parameter has elapsed, whichever happens first.

Thread Blocking

The operation `wait_for_historical_data` blocks the calling thread until either all "historical" data is received, or the duration specified by the `max_wait` parameter elapses, whichever happens first. A return value of `RETCODE_OK` indicates that all the "historical" data was received; a return value of `RETCODE_TIMEOUT` indicates that `max_wait` elapsed before all the data was received.

Return Code

When the operation returns:

- *RETCODE_OK* - the "historical" data is received
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `DataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `DataReader` is not enabled.
- *RETCODE_TIMEOUT* - not all data is received before `max_wait` elapsed.

3.5.2.45 **wait_for_historical_data_w_condition**

Scope

```
DDS.DataReader
```

Synopsis

```
import DDS.*;
int
   wait_for_historical_data_w_condition
      (String filter_expression,
       String[] filter_parameters,
       Time_t min_source_timestamp,
       Time_t max_source_timestamp
       ResourceLimitsQosPolicy resource_limits,
       Duration_t max_wait)
```

Description

This operation will block the application thread until all historical data that matches the supplied conditions is received.

⚠ **NOTE:** This operation only makes sense when the receiving node has configured its durability service as an On_Request alignee. (See also the description of the `OpenSplice/DurabilityService/NameSpaces/Policy[@alignee]` attribute in the *Deployment Guide*.) Otherwise the Durability Service will not distinguish between separate reader requests and still inject the full historical data set in each reader.

Additionally, when creating the DataReader, the DurabilityQos.kind of the DataReaderQos needs to be set to VOLATILE, to ensure that historical data that potentially is available already at creation time is not immediately delivered to the DataReader at that time.

Parameters

- **in** `String filter_expression` - the SQL expression (subset of SQL), which defines the filtering criteria (NULL when no SQL filtering is needed).

- **in** `String[] filter_parameters` – sequence of strings with the parameter values used in the SQL expression (*i.e.*, the number of `%n` tokens in the expression). The number of values in `expression_parameters` must be equal to or greater than the highest referenced `%n` token in the `filter_expression` (*e.g.* if `%1` and `%8` are used as parameters in the `filter_expression`, the `expression_parameters` should contain at least $n + 1 = 9$ values).

- **in** `Time_t min_source_timestamp` - Filter out all data published before this time. The special constant `TIMESTAMP_INVALID` can be used when no minimum filter is needed. The value of `min_source_timestamp.sec` must be less than `0x7fffffff` otherwise it will be recognized as `TIMESTAMP_INVALID_SEC`.

- **in** `Time_t max_source_timestamp` - Filter out all data published after this time. The special constant `TIMESTAMP_INVALID` can be used when no maximum filter is needed. The value of `max_source_timestamp.sec` must be less than `0x7fffffff` otherwise it will be recognized as `TIMESTAMP_INVALID_SEC`.

- **in** `ResourceLimitsQosPolicy resource_limits` - Specifies limits on the maximum amount of historical data that may be received.

- **in** `Duration_t max_wait` - The maximum duration the application thread is blocked during this operation.

PRISMTECH

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_PRECONDITION_NOT_MET, RETCODE_ALREADY_DELETED, RETCODE_NOT_ENABLED, RETCODE_TIMEOUT.

Detailed Description

This operation is similar to the *wait_for_historical_data* operation, but instead of inserting all historical data into the DataReader, only data that matches the conditions expressed by the parameters to this opération is inserted. For more information about historical data please refer to section *3.5.2.44* on page 406.

By using filter_expression and filter_parameters, data can be selected or discarded based on content. The filter_expression must adhere to SQL syntax of the *WHERE* clause as described in Appendix H, *DCPS Queries and Filters*. Constraints on the age of data can be set by using the min_source_timestamp and max_source_timestamp parameters. Only data published within this timeframe will be selected. Note that TIMESTAMP_INVALID is also accepted as a lower or upper timeframe limit. The amount of selected data can be further reduced by the resource_limits parameter. This QosPolicy allows to set a limit on the number of samples, instances and samples per instance that are to be received.

Return Code

When the operation returns:

- *RETCODE_OK* - the historical data is received.
- *RETCODE_ERROR* - an internal error occurred.
- *RETCODE_BAD_PARAMETER* - any of the parameters is invalid, including resource_limits that do not meet constraints set on the DataReader.
- *RETCODE_PRECONDITION_NOT_MET* - No Durability service is available or a different request for historical data is already being processed.
- *RETCODE_ALREADY_DELETED* - the DataReader is already deleted.
- *RETCODE_NOT_ENABLED* - the DataReader is not enabled.
- *RETCODE_TIMEOUT* - not all data is received before max_wait elapsed.

3.5.2.46 Interface FooDataReader

The pre-processor generates from IDL type descriptions the application <type>DataReader interfaces. For each application data type that is used as Topic data type, a typed interface <type>DataReader extends the DataReader

interface. In this paragraph, the interface `FooDataReader` in the package `SPACE` describes the operations of these extending `<type>DataReader` interfaces as an example for the fictional application type `Foo` (defined in the module `SPACE`).

State Masks

A `FooDataReader` refers to exactly one `TopicDescription` (either a `Topic`, a `ContentFilteredTopic` or a `MultiTopic`) that identifies the data to be read. Therefore it refers to exactly one data type. The `Topic` must exist prior to the `FooDataReader` creation. The `FooDataReader` may give access to several instances of the data type, which are distinguished from each other by their `key`. The `FooDataReader` is attached to exactly one `Subscriber` which acts as a factory for it.

The interface description is as follows:

```
 public interface FooDataReader
{
//
// extends interface Entity
//
// public StatusCondition
//     get_statuscondition
//         (void);
// public int
//     get_status_changes
//         (void);
// public int
//     enable
//         (void);
//
// extended to interface DataReader
//
// public ReadCondition
//     create_readcondition
//         (int sample_states,
//         int view_states,
//         int instance_states);

// public QueryCondition
//     create_querycondition
//         (int sample_states,
//         int view_states,
//         int instance_states,
//         String query_expression,
//         String[] query_parameters);

// public int
//     delete_readcondition
//         (ReadCondition a_condition);
```

PRISMTECH

```
// public int
//    delete_contained_entities
//        (void);

// public int
//    set_qos
//        (DataReaderQos qos);

// public int
//    get_qos
//        (DataReaderQosHolder qos);

// public int
//    set_listener
//        (DataReaderListener a_listener,
//            int mask);

// public DataReaderListener
//    get_listener
//        (void);

// public TopicDescription
//    get_topicdescription
//        (void);

// public Subscriber
//    get_subscriber
//        (void);

// public int
//    get_sample_rejected_status
//        (SampleRejectedStatusHolder status);

// public int
//    get_liveliness_changed_status
//        (LivelinessChangedStatusHolder status);

// public int
//    get_requested_deadline_missed_status
//        (RequestedDeadlineMissedStatusHolder status);

// public int
//    get_requested_incompatible_qos_status
//        (RequestedIncompatibleQosStatusHolder status);

// public int
//    get_subscription_matched_status
//        (SubscriptionMatchedStatusHolder status);
```

```
                // public int
                //    get_sample_lost_status
                //        (SampleLostStatusHolder status);

                // public int
                //    wait_for_historical_data
                //        (Duration_t max_wait);

                // public int
                //    get_matched_publications
                //        (InstanceHandleSeqHolder publication_handles);

                // public int
                //    get_matched_publication_data
                //        (PublicationBuiltinTopicDataHolder publication_data,
                //        long publication_handle);
                //
                // implemented API operations
                //
                    public int
                        read
                            (FooSeqHolder data_values,
                              SampleInfoSeqHolder info_seq,
                              int max_samples,
                              int sample_states,
                              int view_states,
                              int instance_states);
                    public int
                        take
                            (FooSeqHolder data_values,
                              SampleInfoSeqHolder info_seq,
                              int max_samples,
                              int sample_states,
                              int view_states,
                              int instance_states);
                    public int
                        read_w_condition
                            (FooSeqHolder data_values,
                              SampleInfoSeqHolder info_seq,
                              int max_samples,
                              ReadCondition a_condition);
                    public int
                        take_w_condition
                            (FooSeqHolder data_values,
                              SampleInfoSeqHolder info_seq,
                              int max_samples,
                              ReadCondition a_condition);
                    public int
                        read_next_sample
                            (FooHolder data_value,
```

```
                        SampleInfoHolder sample_info);
               public int
                  take_next_sample
                     (FooHolder data_value,
                     SampleInfoHolder sample_info);
               public int
                  read_instance
                     (FooSeqHolder data_values,
                       SampleInfoSeqHolder info_seq,
                       int max_samples,
                       long a_handle,
                       int sample_states,
                       int view_states,
                       int instance_states);
               public int
                  take_instance
                     (FooSeqHolder data_values,
                       SampleInfoSeqHolder info_seq,
                       int max_samples,
                       long a_handle,
                       int sample_states,
                       int view_states,
                       int instance_states);
               public int
                  read_next_instance
                     (FooSeqHolder data_values,
                       SampleInfoSeqHolder sample_info,
                       int max_samples,
                       long a_handle,
                       int sample_states,
                       int view_states,
                       int instance_states);
               public int
                  take_next_instance
                     (FooSeqHolder data_values,
                       SampleInfoSeqHolder sample_info,
                       int max_samples,
                       long a_handle,
                       int sample_states,
                       int view_states,
                       int instance_states);
               public int
                  read_next_instance_w_condition
                     (FooSeqHolder data_values,
                       SampleInfoSeqHolder info_seq,
                       int max_samples,
                       long a_handle,
                       ReadCondition a_condition);
               public int
                  take_next_instance_w_condition
```

```
                (FooSeqHolder data_values,
                  SampleInfoSeqHolder info_seq,
                  int max_samples,
                  long a_handle,
                  ReadCondition a_condition);
          public int
            return_loan
                (FooSeqHolder data_values,
                  SampleInfoSeqHolder info_seq);
          public int
            get_key_value
              (FooHolder key_holder
                  long handle);
          long
            lookup_instance
                (Foo instance_data);
      };
```

The following paragraphs describe the usage of all `FooDataReader` operations.
The inherited operations are listed but not fully described because they are not
implemented in this class. The full description of these operations is given in the
classes from which they are inherited.

3.5.2.47 create_querycondition (inherited)

This operation is inherited and therefore not described here. See the interface
`DataReader` for further explanation.

Synopsis

```
QueryCondition
   create_querycondition
      (int sample_states,
        int view_states,
        int instance_states,
        String query_expression,
        String[] query_parameters);
```

3.5.2.48 create_readcondition (inherited)

This operation is inherited and therefore not described here. See the interface
`DataReader` for further explanation.

Synopsis

```
ReadCondition
   create_readcondition
      (int sample_states,
       int view_states,
       int instance_states);
```

3.5.2.49 **delete_contained_entities (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
int
   delete_contained_entities
      (void);
```

3.5.2.50 **delete_readcondition (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
   delete_readcondition
      (ReadCondition a_condition);
```

3.5.2.51 **enable (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
public int
   enable
      (void);
```

3.5.2.52 **get_key_value**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
public int
   get_key_value
      (FooHolder key_holder,
        long handle);
```

Description

This operation retrieves the key value of a specific instance.

Parameters

inout FooHolder key_holder - a reference to the sample in which the key values are stored.

in long handle - the handle to the instance from which to get the key value.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation retrieves the key value of the instance referenced to by instance_handle. When the operation is called with an HANDLE_NIL constant as an instance_handle, the operation will return RETCODE_BAD_PARAMETER. The operation will only fill the fields that form the key inside the key_holder instance. This means that the non-key fields are not applicable and may contain garbage.

The operation must only be called on registered instances. Otherwise the operation returns the error RETCODE_PRECONDITION_NOT_MET.

Return Code

When the operation returns:

- *RETCODE_OK* - the key_holder instance contains the key values of the instance;
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - handle is not a valid handle
- *RETCODE_ALREADY_DELETED* - the FooDataReader has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the FooDataReader is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - this instance is not registered.

3.5.2.53 get_listener (inherited)

This operation is inherited and therefore not described here. See the interface DataReader for further explanation.

Synopsis

```
import DDS.*;
public DataReaderListener
```

```
get_listener
    (void);
```

3.5.2.54 get_liveliness_changed_status (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_liveliness_changed_status
        (LivelinessChangedStatusHolder status);
```

3.5.2.55 get_matched_publication_data (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_matched_publication_data
        (PublicationBuiltinTopicDataHolder publication_data,
         long publication_handle);
```

3.5.2.56 get_matched_publications (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_matched_publications
        (InstanceHandleSeqHolder publication_handles);
```

3.5.2.57 get_qos (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public void
    get_qos
        (DataReaderQosHolder qos);
```

3.5.2.58 **get_requested_deadline_missed_status (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
  get_requested_deadline_missed_status
    (RequestedDeadlineMissedStatusHolder status);
```

3.5.2.59 **get_requested_incompatible_qos_status (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
  get_requested_incompatible_qos_status
    (RequestedIncompatibleQosStatusHolder status);
```

3.5.2.60 **get_sample_lost_status (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
  get_sample_lost_status
    (SampleLostStatusHolder status);
```

3.5.2.61 **get_sample_rejected_status (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
  get_sample_rejected_status
    (SampleRejectedStatusHolder status);
```

3.5.2.62 **get_status_changes (inherited)**

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
public int
    get_status_changes
        (void);
```

3.5.2.63 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the interface `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

3.5.2.64 get_subscriber (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public Subscriber
    get_subscriber
        (void);
```

3.5.2.65 get_subscription_match_status (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_subscription_matched_status
        (SubscriptionMatchedStatusHolder status);
```

3.5.2.66 get_topicdescription (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public TopicDescription
    get_topicdescription
        (void);
```

3.5.2.67 **lookup_instance**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public long
   lookup_instance
      (Foo instance_data);
```

Description

This operation returns the value of the instance handle which corresponds to the `instance_data`.

Parameters

in Foo instance_data - the instance for which the corresponding instance handle needs to be looked up.

Return Value

long - Result value is the instance handle which corresponds to the `instance_data`.

Detailed Description

This operation returns the value of the instance handle which corresponds to the `instance_data`. The instance handle can be used in read operations that operate on a specific instance. Note that `DataReader` instance handles are local, and are not interchangeable with `DataWriter` instance handles nor with instance handles of an other `DataReader`. If the `DataReader` is already deleted, the handle value `HANDLE_NIL` is returned.

3.5.2.68 **read**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   read
      (FooSeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       int sample_states,
```

```
            int view_states,
            int instance_states);
```

Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

in int instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`. The data is returned by the parameters `data_values` and `info_seq`. The number of samples that is returned is limited by the parameter `max_samples`. This operation is part of the specialized interface which is generated for the particular application data type (in this case type `Foo`) that is being read. If the `FooDataReader` has no samples that meet the constraints, the return value is `RETCODE_NO_DATA`.

State Masks

The `read` operation depends on a selection of the samples by using three masks:

• `sample_states` is the mask, which selects only those samples with the desired sample states `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both

- `view_states` is the mask, which selects only those samples with the desired view states `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE` or both

- `instance_states` is the mask, which selects only those samples with the desired instance states `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these

Destination Order

In any case, the relative order between the samples of one instance is consistent with the `DestinationOrderQosPolicy` of the `Subscriber`.

When the `DestinationOrderQosPolicy` kind is `BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS`, the samples belonging to the same instances will appear in the relative order in which they were received (FIFO);

When the `DestinationOrderQosPolicy` kind is `BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS`, the samples belonging to the same instances will appear in the relative order implied by the source_timestamp.

Data Sample

In addition to the sample sequence (`data_values` parameter), the operation also returns a sequence of `SampleInfo` objects with the parameter `info_seq`. The `info_seq` objects and `data_values` also determine the behaviour of this operation.

Resource Control

The initial length of the `data_values` and `info_seq` sequences (`received_data.value.length` and `info_seq.value.length`) determine the precise behaviour of the `read` operation. The behaviour of the `read` operation is as specified by the following rules:

- On successful output, the sequence holders contain arrays whose `length` is equal to the number of returned samples. These arrays may be different from the ones originally passed in the sequence holders.

- If the Holder objects have `value == null`, or if their `value` fields point to arrays that have `length == 0`, the `received_data` and `info_seq` sequence are filled with elements that are "loaned" by the `FooDataReader`. On output, the sequence holders contain arrays whose `length` is equal to the number of returned samples. In this case the application will need to "return the loan" to the Data Distribution Service using the `return_loan` operation.

- If the Holder objects have `value` fields that point to arrays that have `length > 0`, the read operation will copy the `Foo` samples and `info_seq` values into the elements already allocated inside the sequences (effectively overwriting the current state of these elements). On output the sequence holders contain arrays whose `length` is equal to the number of returned samples. The application can pre-allocate the elements of the array and does not need to "return the loan". The number of samples copied depends on the relative values of `length` and `max_samples`:

  - If `max_samples == LENGTH_UNLIMITED`, at most `length` values are copied. The use of this variant lets the application limit the number of samples returned to what the sequence can accommodate;

  - If `max_samples <= length`, at most `max_samples` values are copied. The use of this variant lets the application limit the number of samples returned to fewer than what the sequence can accommodate;

  - If `max_samples > length` the read operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`. 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 `FooDataReader`, because the output sequence can not accommodate them.

Buffer Loan

As described above, upon return the `data_values` and `info_seq` sequences may contain elements "loaned" from the Data Distribution Service. If this is the case, the application will need to use the `return_loan` operation to return the "loan" once it is no longer using the data in the sequence. Upon return from `return_loan`, the Holder objects have their `value` set to `null`.

The application must remember if it is necessary to "return the loan" or not. However, in many cases it may be simpler to always call `return_loan`, as this operation is harmless if the sequence does not hold a loan.

Data Sequence

On output, the sequence of data values and the sequence of `SampleInfo` objects are of the same length and are in an one-to-one correspondence. Each `SampleInfo` object provides information, such as the `source_timestamp`, the `sample_state`, `view_state`, and `instance_state`, etc., about the matching sample.

Some elements in the returned sequence may not have valid data:the `valid_data` field in the `SampleInfo` indicates whether the corresponding data value contains any meaningful data. If not, the data value is just a 'dummy' sample for which only

the keyfields have been assigned. It is used to accompany the `SampleInfo` that communicates a change in the `instance_state` of an instance for which there is no 'real' sample available.

For example, when an application always 'takes' all available samples of a particular instance, there is no sample available to report the disposal of that instance. In such a case the `DataReader` will insert a dummy sample into the `data_values` sequence to accompany the `SampleInfo` element in the `info_seq` sequence that communicates the disposal of the instance.

The act of reading a sample sets its sample_state to `READ_SAMPLE_STATE`. If the sample belongs to the most recent generation of the instance, it also sets the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It does not affect the `instance_state` of the instance.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - one or more of the `received_data` and `info_seq` parameters is an invalid reference.
- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.69 read_instance

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   read_instance
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
```

```
            int sample_states,
            int view_states,
            int instance_states);
```

Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in long a_handle - the single instance, the samples belong to.

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

in int instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`. The behaviour is identical to `read` except for that all samples returned belong to the single specified instance whose handle is `a_handle`. Upon successful return, the data collection will contain samples all belonging to the same instance. The data is returned by the parameters `data_values` and `info_seq`. The corresponding `SampleInfo.instance_handle` in `info_seq` will have the value

of a_handle. The DataReader will check that each sample belongs to the specified instance (indicated by a_handle) otherwise it will not place the sample in the returned collection.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - one or more of the received_data and info_seq parameters is an invalid reference or a_handle is not a valid handle.
- *RETCODE_ALREADY_DELETED* - the FooDataReader has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the FooDataReader is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* -one of the following is true:
  - the max_samples > length and max_samples is not LENGTH_UNLIMITED
  - the handle ==HANDLE_NIL.
  - the handle has not been registered with this DataReader.
- RETCODE_NO_DATA - no samples that meet the constraints are available.

3.5.2.70 read_next_instance

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   read_next_instance
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        int sample_states,
        int view_states,
        int instance_states);
```

Description

This operation reads a sequence of Foo samples of the next single instance from the FooDataReader.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in long a_handle - the current single instance, the returned samples belong to the next single instance.

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

in int instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`. The behaviour is similar to `read_instance` (all samples returned belong to a single instance) except that the actual instance is not directly specified. Rather the samples will all belong to the 'next' instance with instance_handle 'greater' (according to some internal-defined order) than a_handle, that has available samples. The data is returned by the parameters `data_values` and `info_seq`. The corresponding `SampleInfo.instance_handle` in `info_seq` will has the value of the next instance with respect to a_handle.

Instance Order

The internal-defined order is not important and is implementation specific. The important thing is that, according to the Data Distribution Service, all instances are ordered relative to each other. This ordering is between the instances, that is, it does not depend on the actual samples received. For the purposes of this explanation it is *'as if'* each instance handle was represented as a unique integer.

The behaviour of `read_next_instance` is *'as if'* the `DataReader` invoked `read_instance` passing the smallest `instance_handle` among all the ones that:

- are greater than `a_handle`
- have available samples (i.e. samples that meet the constraints imposed by the specified states)

The special value `HANDLE_NIL` is guaranteed to be 'less than' any valid `instance_handle`. So the use of the parameter value `a_handle==HANDLE_NIL.value` will return the samples for the instance which has the smallest `instance_handle` among all the instances that contains available samples.

Typical Use

The operation `read_next_instance` is intended to be used in an application-driven iteration where the application starts by passing `a_handle==HANDLE_NIL.value`, examines the samples returned, and then uses the `instance_handle` returned in the `SampleInfo` as the value of `a_handle` argument to the next call to `read_next_instance`. The iteration continues until `read_next_instance` returns the return value `RETCODE_NO_DATA`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - one or more of the `received_data` and `info_seq` parameters is an invalid reference or `a_handle` is not a valid handle.
- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* -one of the following is true:
  - the `max_samples` > `length` and `max_samples` is not `LENGTH_UNLIMITED`
  - the handle has not been registered with this DataReader.

PRISMTECH

- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.71 **read_next_instance_w_condition**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   read_next_instance_w_condition
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        ReadCondition a_condition);
```

Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in long a_handle - the current single instance, the returned samples belong to the next single instance.

in ReadCondition a_condition - a reference to a `ReadCondition` object or `QueryCondition` object which filters the data before it is returned by the read operation.

Return Value

int - Possible return codes of the operation are:
```
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,
RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or
RETCODE_NO_DATA.
```

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition`. The behaviour is identical to `FooDataReader_read_next_instance` except for that the `samples` are filtered by a `ReadCondition` or `QueryCondition`. When using a `ReadCondition`, the result is the same as the `FooDataReader_read_next_instance` operation with the same state parameters filled in as for the `create_readcondition`. In this way, the application can avoid repeating the same parameters, specified when creating the `ReadCondition`. When using a `QueryCondition`, a content based filtering can be done. When either using a `ReadCondition` or `QueryCondition`, the condition must be created by this `FooDataReader`. Otherwise the operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - one or more of the `received_data, info_seq` and `a_condition` parameters is an invalid reference or `a_handle` is not a valid handle.
- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* -one of the following is true:
  - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
  - the handle has not been registered with this DataReader.
- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.72 read_next_sample

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   read_next_sample
```

```
           (FooHolder data_value,
            SampleInfoHolder sample_info);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.73 **read_w_condition**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   read_w_condition
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        ReadCondition a_condition);
```

Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in ReadCondition a_condition - a reference to a `ReadCondition` or `QueryCondition` which filters the data before it is returned by the `read` operation.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition`. The condition reference from both `create_readcondition` or `create_querycondition` may be used. The behaviour is identical to `read` except for that the samples are filtered by a `ReadCondition` or `QueryCondition`. When using a `ReadCondition`, the result is the same as the `read` operation with the same state parameters filled in as for the `create_readcondition`. In this way, the application can avoid repeating the same parameters, specified when creating the `ReadCondition`. When using a `QueryCondition`, a content based filtering can be done. When either using a `ReadCondition` or `QueryCondition`, the condition must be created by this `FooDataReader`. Otherwise the operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - one or more of the `received_data`, `info_seq` and `a_condition` parameters is an invalid reference.
- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.74 **return_loan**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   return_loan
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq);
```

Description

This operation indicates to the `DataReader` that the application is done accessing the sequence of `data_values` and `info_seq`.

Parameters

inout FooSeqHolder data_values - the sample data sequence which was loaned from the `DataReader`.

inout SampleInfoSeqHolder info_seq - the `SampleInfo` object sequence which was loaned from the `DataReader`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation indicates to the `DataReader` that the application is done accessing the sequence of `data_values` and `info_seq` obtained by some earlier invocation of the operation `read` or `take` (or any of the similar operations) on the `DataReader`.

The `data_values` and `info_seq` must belong to a single related pair; that is, they should correspond to a pair returned from a single call to the operation `read` or `take`. The `data_values` and `info_seq` must also have been obtained from the same `DataReader` to which they are returned. If either of these conditions is not met the operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`.

Buffer Loan

The operation `return_loan` allows implementations of the `read` and `take` operations to "loan" buffers from the Data Distribution Service to the application and in this manner provide "zero-copy" access to the data. During the loan, the Data Distribution Service will guarantee that the `data_values` and `info_seq` are not modified.

It is not necessary for an application to return the loans immediately after calling the operation `read` or `take`. However, as these buffers correspond to internal resources inside the `DataReader`, the application should not retain them indefinitely.

Calling return_loan

The use of the `return_loan` operation is only necessary if the call to the operation `read` or `take` "loaned" buffers to the application. This only occurs if the `data_values` and `info_seq` sequences had `length=0` at the time the operation `read` or `take` was called. The application must remember if it is necessary to "return the loan" or not. However, calling the operation `return_loan` on a pair of sequences that does not have a loan is safe and has no side effects.

If the pair of sequences had a loan, upon return from the operation `return_loan` their Holder objects will have their `value` set to `null`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `DataReader` is informed that the sequences will not be used any more

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - either or both of `received_data` or `info_seq` is an invalid reference

- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - one of the following is true

  - the `received_data` and `info_seq` does not belong to a single related pair

  - the `received_data` and `info_seq` was not obtained from this `FooDataReader`

3.5.2.75 set_listener (inherited)

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
  set_listener
    (DataReaderListener a_listener,
      int mask);
```

PRISMTECH

3.5.2.76 **set_qos (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReader` for further explanation.

Synopsis

```
import DDS.*;
public int
   set_qos
       (DataReaderQos qos);
```

3.5.2.77 **take**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   take
       (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
```

Description

This operation reads a sequence of `Foo` samples from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

`in int instance_states` - a mask, which selects only those samples with the desired instance states.

Return Value

`int` - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
   RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,
   RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or
   RETCODE_NO_DATA.

Detailed Description

This operation reads a sequence of `Foo` samples from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available and removed from the `FooDataReader`

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - either or both of `received_data` or `info_seq` is an invalid reference

- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - the `max_samples > length` and `max_samples` is not LENGTH_UNLIMITED

- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.78 take_instance

Scope

      SPACE.FooDataReader

Synopsis

```
import DDS.*;
public int
   take_instance
```

```
          (FooSeqHolder data_values,
            SampleInfoSeqHolder info_seq,
            int max_samples,
            long a_handle,
            int sample_states,
            int view_states,
            int instance_states);
```

Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in long a_handle - the single instance, the samples belong to.

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

in int instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_instance` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available and removed from the `FooDataReader`

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - either or both of `received_data` or `info_seq` is an invalid reference or `a_handle` is not a valid handle.

- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* -one of the following is true:

  - the `max_samples` > `length` and `max_samples` is not `LENGTH_UNLIMITED`

  - the handle `==HANDLE_NIL`.

  - the handle has not been registered with this DataReader.

- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.79 take_next_instance

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
    take_next_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder sample_info,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in long a_handle - the current single instance, the returned samples belong to the next single instance.

in int sample_states - a mask, which selects only those samples with the desired sample states.

in int view_states - a mask, which selects only those samples with the desired view states.

in int instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_next_instance` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available and removed from the `FooDataReader`.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - either or both of `received_data` or `info_seq` is an invalid reference or `a_handle` is not a valid handle.

- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - one of the following is true:
    - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
    - the handle has not been registered with this DataReader.
- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.80 **take_next_instance_w_condition**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   take_next_instance_w_condition
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        ReadCondition a_condition);
```

Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in long a_handle - the current single instance, the returned samples belong to the next single instance.

in ReadCondition a_condition - a reference to a `ReadCondition` or `QueryCondition` which filters the data before it is returned by the read operation.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader, filtered by a ReadCondition or QueryCondition` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_next_instance_w_condition` except for that the `samples` are removed from the `FooDataReader`.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available and removed from the `FooDataReader`.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - one or more of the `received_data, info_seq` and `a_condition` parameters is an invalid reference or `a_handle` is not a valid handle.

- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* -one of the following is true:

  - the `max_samples` > `length` and `max_samples` is not `LENGTH_UNLIMITED`

  - the handle has not been registered with this DataReader.

- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.81 **take_next_sample**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   take_next_sample
      (FooHolder data_value,
       SampleInfoHolder sample_info);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.82 **take_w_condition**

Scope

```
SPACE.FooDataReader
```

Synopsis

```
import DDS.*;
public int
   take_w_condition
      (FooSeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       ReadCondition a_condition);
```

Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeqHolder data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeqHolder info_seq - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in int max_samples - the maximum number of samples that is returned.

in ReadCondition a_condition - a reference to a `ReadCondition` or `QueryCondition` which filters the data before it is returned by the `read` operation.

Return Value

int - Possible return codes of the operation are:
RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,
RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES,
RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or
RETCODE_NO_DATA.

Detailed Description

This operation reads a sequence of Foo samples from the FooDataReader, filtered by a ReadCondition or QueryCondition and by doing so, removes the data from the FooDataReader, so it can not be read or taken again. The behaviour is identical to read_w_condition except for that the samples are removed from the FooDataReader.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available and removed from the FooDataReader.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - one or more of the received_data, info_seq and a_condition parameters is an invalid reference.

- *RETCODE_ALREADY_DELETED* - the FooDataReader has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the FooDataReader is not enabled.

- *RETCODE_PRECONDITION_NOT_MET* - the max_samples > length and max_samples is not LENGTH_UNLIMITED.

- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.83 wait_for_historical_data (inherited)

This operation is inherited and therefore not described here. See the interface DataReader for further explanation.

Synopsis

```
import DDS.*;
public int
   wait_for_historical_data
       (Duration_t max_wait);
```

3.5.2.84 **wait_for_historical_data_w_condition (inherited)**

This operation is inherited and therefore not described here. See the class DataReader for further explanation.

Synopsis

```
import DDS.*;
int
    wait_for_historical_data_w_condition
        (String filter_expression,
         String[] filter_parameters,
         Time_t min_source_timestamp,
         Time_t max_source_timestamp
         ResourceLimitsQosPolicy resource_limits,
         Duration_t max_wait)
```

3.5.3 **Class DataSample**

A DataSample represents an atom of data information (i.e. one value for an instance) as returned by the DataReader's read/take operations. It consists of two parts: A SampleInfo and the Data itself. The Data part is the data as produced by a Publisher. The SampleInfo part contains additional information related to the data provided by the Data Distribution Service.

3.5.4 **Class SampleInfo**

The class SampleInfo represents the additional information that accompanies the data in each sample that is read or taken.

The interface description of this class is as follows:

```
public class SampleInfo
{
    public int sample_state;
    public int view_state;
    public int instance_state;
    public Time_t source_timestamp;
    public long instance_handle;
    public long publication_handle;
    public int disposed_generation_count;
    public int no_writers_generation_count;
    public int sample_rank;
    public int generation_rank;
    public int absolute_generation_rank;
    public boolean valid_data;
    public Time_t reception_timestamp;
};
```

The next paragraph describes the usage of the SampleInfo struct.

3.5.4.1 **SampleInfo**

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class SampleInfo
   {
     public int sample_state;
     public int view_state;
     public int instance_state;
     public Time_t source_timestamp;
     public long instance_handle;
     public long publication_handle;
     public int disposed_generation_count;
     public int no_writers_generation_count;
     public int sample_rank;
     public int generation_rank;
     public int absolute_generation_rank;
     public boolean valid_data;
     public Time_t reception_timestamp;
   };
```

Description

The class `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

Attributes

`int sample_state` - whether or not the corresponding data sample has already been read.

`int view_state` - whether the `DataReader` has already seen samples of the most-current generation of the related instance.

`int instance_state` - whether the instance is alive, has no writers or is disposed of.

`Time_t source_timestamp` - the time provided by the `DataWriter` when the sample was written.

`long instance_handle` - the handle that identifies locally the corresponding instance.

long publication_handle - the handle that identifies locally the `DataWriter` that modified the instance. In fact it is an `instance_handle` of the built-in `DCPSPublication` sample that describes this `DataWriter`. It can be used as a parameter to the `DataReader` operation `get_matched_publication_data` to obtain this built-in `DCPSPublication` sample.

int disposed_generation_count - the number of times the instance has become alive after it was disposed of explicitly by a `DataWriter`.

int no_writers_generation_count - the number of times the instance has become alive after it was disposed of because there were no `DataWriter` objects.

int sample_rank - the number of samples related to the same instance that are found in the collection returned by a `read` or `take` operation.

int generation_rank - the generation difference between the time the sample was received and the time the most recent sample in the collection was received.

int absolute_generation_rank - the generation difference between the time the sample was received and the time the most recent sample was received.

boolean valid_data - whether the DataSample contains any meaningful data. If not, the sample is only used to communicate a change in the `instance_state` of the instance.

Time_t reception_timestamp - the time provided by the `DataReader` when the sample was received.

Detailed Description

The class `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

Generations

A generation is defined as: 'the number of times an instance has become alive (with `instance_state==ALIVE_INSTANCE_STATE`) at the time the sample was received'. Note that the generation counters are initialized to zero when a `DataReader` first detects a never-seen-before instance.

Two types of generations are distinguished: `disposed_generation_count` and `no_writers_generation_count`.

After a `DataWriter` disposes an instance, the `disposed_generation_count` for all `DataReaders` that already knew that instance will be incremented the next time the instance is written again.

If the `DataReader` detects that there are no live `DataWriter` entities, the `instance_state` of the `sample_info` will change from `ALIVE_INSTANCE_STATE` to `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`. The next time the instance is written, `no_writers_generation_count` will be incremented.

Sample Information

`SampleInfo` is the additional information that accompanies the data in each sample that is 'read' or 'taken'. It contains the following information:

- `sample_state` (`READ_SAMPLE_STATE` or `NOT_READ_SAMPLE_STATE`) indicates whether or not the corresponding data sample has already been read.

- `view_state` (`NEW_VIEW_STATE` or `NOT_NEW_VIEW_STATE`) indicates whether the `DataReader` has already seen samples of the most-current generation of the related instance.

- `instance_state` (`ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`) indicates whether the instance is alive, has no writers or if it has been disposed of:

  - `ALIVE_INSTANCE_STATE` if this instance is currently in existence.

  - `NOT_ALIVE_DISPOSED_INSTANCE_STATE` if this instance was disposed of by a `DataWriter`.

  - `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` none of the `DataWriter` objects currently "alive" (according to the `LivelinessQosPolicy`) are writing the instance.

- `source_timestamp` indicates the time provided by the `DataWriter` when the sample was written.

- `instance_handle` indicates locally the corresponding instance.

- `publication_handle` is used by the DDS implementation to locally identify the corresponding source DataWriter. You can access more detailed information about this particular publication by passing its `publication_handle` to either the `get_matched_publication_data` operation on the DataReader or to the `read_instance` operation on the built-in reader for the "DCPSPublication" topic.

⚠ Be aware that since an instance handle is an opaque datatype, it does not necessarily mean that the handle obtained from the `publication_handle` has the same value as the one that appears in the `instance_handle` field of the `SampleInfo` when retrieving the publication info through corresponding "DCPSPublication" built-in reader. You can't just compare two handles to determine whether they represent the same publication. If you want to know

whether two handles actually do represent the same publication, use both handles to retrieve their corresponding `PublicationBuiltinTopicData` samples and then compare the `key` field of both samples.

- `disposed_generation_count` indicates the number of times the instance has become alive after it was disposed of explicitly by a `DataWriter`, at the time the sample was received.

- `no_writers_generation_count` indicates the number of times the instance has become alive after its `instance_state` has been `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`, at the time the sample was received.

- `sample_rank` indicates the number of samples related to the same instance that follow in the collection returned by a `read` or `take` operation.

- `generation_rank` indicates the generation difference (number of times the instance was disposed of 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.

- `absolute_generation_rank` indicates the generation difference (number of times the instance was disposed of 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.

- `valid_data` indicates whether the corresponding data value contains any meaningful data. If not, the data value is just a 'dummy' sample for which only the keyfields have been assigned. It is used to accompany the `SampleInfo` that communicates a change in the `instance_state` of an instance for which there is no 'real' sample available.

- `reception_timestamp` indicates the time provided by the `DataReader` when the sample was inserted.

   ⚠️ **NOTE:** This is an OpenSplice-specific extension to the `SampleInfo` struct and is *not* part of the DDS Specification.

3.5.5 SubscriberListener Interface

Since a `Subscriber` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `SubscriberListener`. This interface must be implemented by the application. A user-defined class must be provided by the application which must implement the `SubscriberListener` interface.

⚠ *All* `SubscriberListener` operations *must* be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The `SubscriberListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `SubscriberListener` is related to changes in communication status.

The interface description is as follows:

```
 public interface SubscriberListener
 {
 //
 // extends interface DataReaderListener
 //
 // public void
 //     on_requested_deadline_missed
 //         (DataReader reader,
 //         RequestedDeadlineMissedStatus status);

 // public void
 //     on_requested_incompatible_qos
 //         (DataReader reader,
 //         RequestedIncompatibleQosStatus status);

 // public void
 //     on_sample_rejected
 //         (DataReader reader,
 //         SampleRejectedStatus status);

 // public void
 //     on_liveliness_changed
 //         (DataReader reader,
 //         LivelinessChangedStatus status);

 // public void
 //     on_data_available
 //         (DataReader reader);

 // public void
 //     on_subscription_matched
 //         (DataReader reader,
 //         SubscriptionMatchedStatus status);

 // public void
 //     on_sample_lost
 //         (DataReader reader,
```

```
//          SampleLostStatus status);
//
//  external operations
//
   public void
      on_data_on_readers
         (Subscriber subs);
//
// implemented API operations
//       <no operations>
//
};
```

The following paragraphs list all `SubscriberListener` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited. The abstract operation is fully described since it must be implemented by the application.

3.5.5.1 **on_data_available (inherited)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_data_available
      (DataReader reader);
```

3.5.5.2 **on_data_on_readers**

Scope

```
DDS.SubscriberListener
```

Synopsis

```
import DDS.*;
public void
   on_data_on_readers
      (Subscriber subs);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when new data is available.

Parameters

in Subscriber subs - contain a reference to the Subscriber for which data is available (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when new data is available for this Subscriber. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant SubscriberListener is installed and enabled for the DATA_ON_READERS_STATUS.

The Data Distribution Service will provide a reference to the Subscriber in the parameter subs for use by the application.

The statuses DATA_ON_READERS_STATUS and DATA_AVAILABLE_STATUS will occur together. In case these status changes occur, the Data Distribution Service will look for an attached and activated SubscriberListener or DomainParticipantListener (in that order) for the DATA_ON_READERS_STATUS. In case the DATA_ON_READERS_STATUS can not be handled, the Data Distribution Service will look for an attached and activated DataReaderListener, SubscriberListener or DomainParticipantListener for the DATA_AVAILABLE_STATUS (in that order).

Note that if on_data_on_readers is called, then the Data Distribution Service will not try to call on_data_available, however, the application can force a call to the callback function on_data_available of DataReaderListener objects that have data by means of the notify_datareaders operation.

3.5.5.3 **on_liveliness_changed (inherited)**

This operation is inherited and therefore not described here. See the interface DataReaderListener for further explanation.

Synopsis

```
import DDS.*;
public void
  on_liveliness_changed
    (DataReader reader,
      LivelinessChangedStatus status);
```

3.5.5.4 on_requested_deadline_missed (inherited)

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_requested_deadline_missed
      (DataReader reader,
        RequestedDeadlineMissedStatus status);
```

3.5.5.5 on_requested_incompatible_qos (inherited)

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
public void]
   on_requested_incompatible_qos
      (DataReader reader,
        RequestedIncompatibleQosStatus status);
```

3.5.5.6 on_sample_lost (inherited)

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_sample_lost
      (DataReader reader,
        SampleLostStatus status);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.5.7 on_sample_rejected (inherited)

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
public void
   on_sample_rejected
      (DataReader reader,
        SampleRejectedStatus status);
```

PRISMTECH

3.5.5.8 **on_subscription_matched (inherited,)**

This operation is inherited and therefore not described here. See the interface `DataReaderListener` for further explanation.

Synopsis

```
import DDS.*;
public void
    on_subscription_matched
        (DataReader reader,
          SubscriptionMatchedStatus status);
```

3.5.6 **DataReaderListener interface**

Since a `DataReader` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `DataReaderListener`. This interface must be implemented by the application. A user-defined class must be provided by the application which must implement the `DataReaderListener` interface. **All** `DataReaderListener` operations **must** be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

⚠️ All operations for this interface must be implemented in the user-defined class, it is up to the application whether an operation is empty or contains some functionality.

The `DataReaderListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `DataReaderListener` is related to changes in communication status.

The interface description of this class is as follows:

```
 public interface DataReaderListener
{
//
// external operations
//
    public void
        on_requested_deadline_missed
            (DataReader reader,
            RequestedDeadlineMissedStatus status);
    public void
        on_requested_incompatible_qos
            (DataReader reader,
            RequestedIncompatibleQosStatus status);

    public void
        on_sample_rejected
```

```
                    (DataReader reader,
                    SampleRejectedStatus status);

            public void
               on_liveliness_changed
                  (DataReader reader,
                  LivelinessChangedStatus status);

            public void
               on_data_available
                  (DataReader reader);

            public public void
               on_subscription_matched
                  (DataReader reader,
                  SubscriptionMatchedStatus status);

            public void
               on_sample_lost
                  (DataReader reader,
                  SampleLostStatus status);
      //
      // implemented API operations
      //      <no operations>
      //
      };
```

The following paragraphs describe the usage of all `DataReaderListener` operations. These abstract operations are fully described because they must be implemented by the application.

3.5.6.1 on_data_available

Scope

```
    DDS.DataReaderListener
```

Synopsis

```
    import DDS.*;
    public void
       on_data_available
          (DataReader reader);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when new data is available.

Parameters

in DataReader reader - contain a reference to the `DataReader` for which data is available (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when new data is available for this `DataReader`. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `DATA_AVAILABLE_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` for use by the application.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` will occur together. In case these status changes occur, the Data Distribution Service will look for an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look for an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

Note that if `on_data_on_readers` is called, then the Data Distribution Service will not try to call `on_data_available`, however, the application can force a call to the `DataReader` objects that have data by means of the `notify_datareaders` operation.

3.5.6.2 **on_liveliness_changed**

Scope

```
DDS.DataReaderListener
```

Synopsis

```
import DDS.*;
public void
   on_liveliness_changed
      (DataReader reader,
        LivelinessChangedStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the liveliness of one or more `DataWriter` objects that were writing instances read through this `DataReader` has changed.

Parameters

in `DataReader reader` - contain a reference to the `DataReader` for which the liveliness of one or more `DataWriter` objects has changed (this is an input to the application provided by the Data Distribution Service).

in `LivelinessChangedStatus status` - contain the `LivelinessChangedStatus` object (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the liveliness of one or more `DataWriter` objects that were writing instances read through this `DataReader` has changed. In other words, some `DataWriter` have become "alive" or "not alive". The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `LIVELINESS_CHANGED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `LivelinessChangedStatus` object for use by the application.

3.5.6.3 on_requested_deadline_missed

Scope

```
DDS.DataReaderListener
```

Synopsis

```
import DDS.*;
public void
   on_requested_deadline_missed
      (DataReader reader,
         RequestedDeadlineMissedStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected.

Parameters

in `DataReader reader` - contain a reference to the `DataReader` for which the deadline was missed (this is an input to the application provided by the Data Distribution Service).

in `RequestedDeadlineMissedStatus status` - contain the `RequestedDeadlineMissedStatus` object (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected for a specific instance. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `REQUESTED_DEADLINE_MISSED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `RequestedDeadlineMissedStatus` object in the parameter `status` for use by the application.

3.5.6.4 **on_requested_incompatible_qos**

Scope

    DDS.DataReaderListener

Synopsis

```
import DDS.*;
public void
   on_requested_incompatible_qos
      (DataReader reader,
        RequestedIncompatibleQosStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `REQUESTED_INCOMPATIBLE_QOS_STATUS` changes.

Parameters

in DataReader reader - a reference to the `DataReader` provided by the Data Distribution Service.

in   RequestedIncompatibleQosStatus   status - the `RequestedIncompatibleQosStatus` object provided by the Data Distribution Service.

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `REQUESTED_INCOMPATIBLE_QOS_STATUS` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `REQUESTED_INCOMPATIBLE_QOS_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `RequestedIncompatibleQosStatus` object in the parameter `status`, for use by the application.

The application can use this operation as a callback function implementing a proper response to the status change. This operation is enabled by setting the `REQUESTED_INCOMPATIBLE_QOS_STATUS` in the `mask` in the call to `DataReader.set_listener`. When the `DataReaderListener` on the `DataReader` is not enabled for the `REQUESTED_INCOMPATIBLE_QOS_STATUS`, the status change will propagate to the `SubscriberListener` of the `Subscriber` (if enabled) or to the `DomainParticipantListener` of the `DomainParticipant` (if enabled).

3.5.6.5 on_sample_lost

Scope

```
DDS.DataReaderListener
```

Synopsis

```
import DDS.*;
public void
   on_sample_lost
      (DataReader reader,
         SampleLostStatus status);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.6.6 on_sample_rejected

Scope

```
DDS.DataReaderListener
```

Synopsis

```
import DDS.*;
public void
   on_sample_rejected
      (DataReader reader,
         SampleRejectedStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when a sample has been rejected.

Parameters

in DataReader reader - contain a reference to the `DataReader` for which a sample has been rejected (this is an input to the application provided by the Data Distribution Service).

in SampleRejectedStatus status - contain the `SampleRejectedStatus` object (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when a (received) sample has been rejected. Samples may be rejected by the `DataReader` when it runs out of `resource_limits` to store incoming samples. Ususally this means that old samples need to be 'consumed' (for example by 'taking' them instead of 'reading' them) to make room for newly incoming samples.

PRISMTECH

The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `SAMPLE_REJECTED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `SampleRejectedStatus` object in the parameter `status` for use by the application.

3.5.6.7 **on_subscription_matched (abstract)**

Scope

```
DDS.DataReaderListener
```

Synopsis

```
import DDS.*;
public void
   on_subscription_matched
      (DataReader reader,
        SubscriptionMatchedStatus status);
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current subscription, or when an existing match has ceased to exist.

Parameters

in DataReader reader - contains a reference to the `DataReader` for which a match has been discovered (this is an input to the application provided by the Data Distribution Service).

in SubscriptionMatchedStatus status - contains the `SubscriptionMatchedStatus` object (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current subscription, or when an existing match has ceased to exist. Usually this means that a new DataWriter that matches the Topic and that has compatible Qos as the current `DataReader` has either been discovered, or that a previously discovered DataWriter has ceased to be matched to the current `DataReader`. A DataWriter may cease to

match when it gets deleted, when it changes its Qos to a value that is incompatible with the current `DataReader` or when either the `DataReader` or the DataWriter has chosen to put its matching counterpart on its ignore-list using the `ignore_publication` or `ignore_subcription` operations on the `DomainParticipant`.

The implementation of this Listener operation may be left empty when this functionality is not needed: it will only be called when the relevant `DataReaderListener` is installed and enabled for the `SUBSCRIPTION_MATCHED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `SubscriptionMatchedStatus` object in the parameter `status` for use by the application.

3.5.7 Interface ReadCondition

The `DataReader` objects can create a set of `ReadCondition` (and `StatusCondition`) objects which provide support (in conjunction with `WaitSet` objects) for an alternative communication style between the Data Distribution Service and the application (i.e., state-based rather than event-based).

`ReadCondition` objects allow an `DataReader` to specify the data samples it is interested in (by specifying the desired sample-states, view-states, and instance-states); see the parameter definitions for `DataReader`'s `create_readcondition` operation. This allows the Data Distribution Service to trigger the condition only when suitable information is available. `ReadCondition` objects are to be used in conjunction with a `WaitSet`. More than one `ReadCondition` may be attached to the same `DataReader`.

The interface description of this class is as follows:

```
 public interface ReadCondition
{
//
// extends to interface Condition
//
// boolean
//     get_trigger_value
//         (void);
//
// implemented API operations
//
    int
       get_sample_state_mask
           (void);

    int
       get_view_state_mask
```

```
                        (void);

            int
                get_instance_state_mask
                    (void);

            DataReader
                get_datareader
                    (void);
        };
```

The following paragraphs describe the usage of all `ReadCondition` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.7.1 get_datareader

Scope

```
DDS.ReadCondition
```

Synopsis

```
import DDS.*;
public DataReader
    get_datareader
        (void);
```

Description

This operation returns the `DataReader` associated with the `ReadCondition`.

Parameters

<none>

Return Value

DataReader - Result value is a reference to the `DataReader`.

Detailed Description

This operation returns the `DataReader` associated with the `ReadCondition`. Note that there is exactly one `DataReader` associated with each `ReadCondition` (i.e. the `DataReader` that created the `ReadCondition` object).

3.5.7.2 get_instance_state_mask

Scope

```
DDS.ReadCondition
```

PRISMTECH

Synopsis

```
import DDS.*;
public int
   get_instance_state_mask
        (void);
```

Description

This operation returns the set of `instance_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

Parameters

<none>

Return Value

int - Result value are the `instance_states` specified when the `ReadCondition` was created.

Detailed Description

This operation returns the set of `instance_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

The `instance_states` returned are the `instance_states` specified when the `ReadCondition` was created. `instance_states` can be `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these.

3.5.7.3 get_sample_state_mask

Scope

```
DDS.ReadCondition
```

Synopsis

```
import DDS.*;
public int
   get_sample_state_mask
        (void);
```

Description

This operation returns the set of `sample_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

Parameters

Return Value

SampleStateMask - Result value are the `sample_states` specified when the `ReadCondition` was created.

Detailed Description

This operation returns the set of `sample_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

The `sample_states` returned are the `sample_states` specified when the `ReadCondition` was created. `sample_states` can be `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both.

3.5.7.4 **get_trigger_value (inherited)**

This operation is inherited and therefore not described here. See the interface `Condition` for further explanation.

Synopsis

```
import DDS.*;
public boolean
   get_trigger_value
      (void);
```

3.5.7.5 **get_view_state_mask**

Scope

```
DDS.ReadCondition
```

Synopsis

```
import DDS.*;
public int
   get_view_state_mask
      (void);
```

Description

This operation returns the set of `view_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

Parameters

<none>

Return Value

ViewStateMask - Result value are the `view_states` specified when the `ReadCondition` was created.

Detailed Description

This operation returns the set of `view_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

The `view_states` returned are the `view_states` specified when the `ReadCondition` was created. `view_states` can be `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE` or both.

3.5.8 Interface QueryCondition

`QueryCondition` objects are specialized `ReadCondition` objects that allow the application to specify a filter on the locally available data. The `DataReader` objects accept a set of `QueryCondition` objects for the `DataReader` and provide support (in conjunction with `WaitSet` objects) for an alternative communication style between the Data Distribution Service and the application (i.e., state-based rather than event-based).

Query Function

`QueryCondition` objects allow an application to specify the data samples it is interested in (by specifying the desired sample-states, view-states, instance-states and query expression); see the parameter definitions for `DataReader`'s `read`/`take` operations. This allows the Data Distribution Service to trigger the condition only when suitable information is available. `QueryCondition` objects are to be used in conjunction with a `WaitSet`. More than one `QueryCondition` may be attached to the same `DataReader`.

The query (`query_expression`) is similar to an SQL WHERE clause and can be parameterised by arguments that are dynamically changeable with the `set_query_parameters` operation.

The interface description is as follows:

```
 public interface QueryCondition
 {
 //
 // extends interface ReadCondition
 //
 // public int
 //     get_sample_state_mask
 //         (void);

 // public int
 //     get_view_state_mask
 //         (void);

 // public int
 //     get_instance_state_mask
 //         (void);
```

```
                // public DataReader
                //    get_datareader
                //        (void);
                // public boolean
                //    get_trigger_value
                //        (void);
                //
                // implemented API operations
                //
                   public String
                      get_query_expression
                          (void);

                   public int
                      get_query_parameters
                          (StringSeqHolder query_parameters);

                   public int
                      set_query_parameters
                          (String[] query_parameters);
                };
```

The following paragraphs describe the usage of all `QueryCondition` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.8.1 **get_datareader (inherited)**

This operation is inherited and therefore not described here. See the interface `ReadCondition` for further explanation.

Synopsis

```
import DDS.*;
public DataReader
   get_datareader
       (void);
```

3.5.8.2 **get_instance_state_mask (inherited)**

This operation is inherited and therefore not described here. See the interface `ReadCondition` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_instance_state_mask
       (void);
```

3.5.8.3 **get_query_parameters**

Scope

```
DDS.QueryCondition
```

Synopsis

```
import DDS.*;
public int
    get_query_parameters
        (StringSeqHolder query_parameters);
```

Description

This operation obtains the `query_parameters` associated with the `QueryCondition`.

Parameters

inout `StringSeqHolder query_parameters` - a reference to the destination `StringSeqHolder` object in which the the parameters used in the SQL expression will be copied.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `query_parameters` associated with the `QueryCondition`. That is, the parameters specified on the last successful call to `set_query_parameters` or, if `set_query_parameters` was never called, the arguments specified when the `QueryCondition` were created.

The resulting Holder contains a sequence of strings with the parameters used in the SQL expression (*i.e.*, the `%n` tokens in the expression). The number of parameters in the result sequence will exactly match the number of `%n` tokens in the query expression associated with the `QueryCondition`.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of query parameters applied to this `QueryCondition` has successfully been copied into the specified `query_parameters` parameter.
- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED* - the `QueryCondition` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.8.4 **get_query_expression**

Scope

```
DDS.QueryCondition
```

Synopsis

```
import DDS.*;
public String
   get_query_expression
      (void);
```

Description

This operation returns the query expression associated with the `QueryCondition`.

Parameters

<none>

Return Value

String - Result value is a reference to the query expression associated with the `QueryCondition`.

Detailed Description

This operation returns the query expression associated with the `QueryCondition`. That is, the expression specified when the `QueryCondition` was created. The operation will return `null` when there was an internal error or when the `QueryCondition` was already deleted. If there were no parameters, an empty sequence is returned.

3.5.8.5 **get_sample_state_mask (inherited)**

This operation is inherited and therefore not described here. See the interface `ReadCondition` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_sample_state_mask
      (void);
```

3.5.8.6 **get_trigger_value (inherited)**

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```
import DDS.*;
public boolean
    get_trigger_value
        (void);
```

3.5.8.7 **get_view_state_mask (inherited)**

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_view_state_mask
        (void);
```

3.5.8.8 **set_query_parameters**

Scope

```
DDS.QueryCondition
```

Synopsis

```
import DDS.*;
public int
    set_query_parameters
        (String[] parameters);
```

Description

This operation changes the query parameters associated with the QueryCondition.

Parameters

in String[] query_parameters - a sequence of strings which are the parameters used in the SQL query string (i.e., the "%n" tokens in the expression).

Return Value

int - Possible return codes of the operation are:
   RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation changes the query parameters associated with the `QueryCondition`. The parameter `query_parameters` is a sequence of strings which are the parameters used in the SQL query string (i.e., the "%n" tokens in the expression). The number of values in `query_parameters` must be equal or greater than the highest referenced %n token in the query_expression (e.g. if `%1` and `%8` are used as parameter in the `query_expression`, the `query_parameters` should at least contain n+1 = 9 values).

Return Code

When the operation returns:

- *RETCODE_OK* - the query parameters associated with the `QueryCondition` are changed.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the number of parameters in `query_parameters` does not match the number of "%n" tokens in the expression for this `QueryCondition` or one of the parameters is an illegal parameter.

- *RETCODE_ALREADY_DELETED* - the `QueryCondition` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.9 Class DataReaderView (abstract)

A DataReaderView allows the application to create an additional view on the dataset stored in a DataReader. The view is expressed by an (optional) alternative key list specified in the `DataReaderViewQos`, which allows it to specify an alternative storage spectrum. Applications might prefer such an alternative storage spectrum (for example by adding or removing key-fields) because it may help them to process the samples in a different order/cohesion than what they will have when they use the original key-list.

A DataReaderView has the following properties:

- Any `DataReaderView` belongs to exactly one `DataReader`.

- A `DataReader` can have zero to many `DataReaderViews` attached (all with their own key_list definitions).

- The `DataReaderView` has the same interface as the `DataReader`, with its read and take variants, including w_condition and next_instance, next_sample, etc. It also supports `ReadConditions` and `QueryConditions` like a `DataReader` does.

- Any sample that is inserted into the DataReader will introduce a corresponding DataViewSample in all its attached DataReaderViews in a ViewInstance as defined by the keys specified in the DataReaderView Qos key_list when the view was created.

- Like samples in a DataReader, DataViewSamples in a DataReaderView belong to exactly one ViewInstance. Instances in the dataReaderView do not have any instance state information though. The instance state information found in the SampleInfo for each DataReaderView sample is copied from the corresponding DataReader sample.

- Whenever a sample is taken from the DataReader, its corresponding samples in all attached DataReaderViews will be removed as well. The same goes for samples that are pushed out of the DataReader instance history (in case of a KEEP_LAST HistoryQosPolicy) or for samples whose lifespan expired.

- A ViewInstance always has an infinite history depth; samples can not be pushed out of the view.

- Whenever a sample is taken from a DataReaderView, it is removed from that DataReaderView but not from the DataReader, nor from any of its other views. If all samples in a ViewInstance are taken, then that ViewInstance is destroyed.

DataReaderView is an abstract class. It is specialized for each particular application data type. For a fictional application data type "Foo" (defined in the module SPACE) the specialized class would be SPACE.FooDataReaderView.

The interface description of this class is as follows:

```
public interface DataReaderView
{
//
// extends interface class Entity
//
// public StatusCondition
//    get_statuscondition
//        (void);
// public int
//    get_status_changes
//        (void);
// public int
//    enable
//        (void);
//
// operations (implemented in data type specific DataReaderView)
//
// public int
//    get_key_value
//        (<data>Holder key_holder,
//         long handle);
```

```
//
// public long
//    lookup_instance
//        (<data> instance_data);
//
// public int
//    read
//        (<data>SeqHolder data_values,
//         SampleInfoSeqHolder info_seq,
//         int max_samples,
//         int sample_states,
//         int view_states,
//         int instance_states);
//
// public int
//    read_instance
//        (<data>SeqHolder data_values,
//         SampleInfoSeqHolder info_seq,
//         int max_samples,
//         long a_handle,
//         int sample_states,
//         int view_states,
//         int instance_states);
//
// public int
//    read_next_instance
//        (<data>SeqHolder data_values,
//         SampleInfoSeqHolder info_seq,
//         int max_samples,
//         long a_handle,
//         int sample_states,
//         int view_states,
//         int instance_states);
//
// public int
//    read_next_instance_w_condition
//        (<data>SeqHolder data_values,
//         SampleInfoSeqHolder info_seq,
//         int max_samples,
//         long a_handle,
//         ReadCondition a_condition);
//
// public int
//    read_next_sample
//        (<data>Holder data_values,
//         SampleInfoHolder sample_info);
//
// public int
//    read_w_condition
//        (<data>SeqHolder data_values,
```

```
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          ReadCondition a_condition);
//
// public int
//    return_loan
//        (<data>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq);
//
// public int
//    take
//        (<data>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          int sample_states,
//          int view_states,
//          int instance_states);
//
// public int
//    take_instance
//        (<data>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
//
// public int
//    take_next_instance
//        (<data>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
//
// public int
//    take_next_instance_w_condition
//        (<data>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          long a_handle,
//          ReadCondition a_condition);
//
// public int
//    take_next_sample
//        (<data>Holder data_values,
//          SampleInfoHolder sample_info);
```

```
            //
            // public int
            //    take_w_condition
            //        (<data>SeqHolder data_values,
            //         SampleInfoSeqHolder info_seq,
            //         int max_samples,
            //         ReadCondition a_condition);
            //
            // implemented API operations

            public QueryCondition
               create_querycondition
                  (int sample_states,
                    int view_states,
                    int instance_states,
                    String query_expression,
                    String[] query_parameters);

            public ReadCondition
               create_readcondition
                  (int sample_states,
                    int view_states,
                    int instance_states);

            public int
               delete_contained_entities
                  (void);

            public int
               delete_readcondition
                  (ReadCondition a_condition);

            public DataReader
               get_datareader
                  (void);

            public int
               get_qos
                  (DataReaderViewQosHolder qos);

            public int
               set_qos
                  (DataReaderViewQos qos);
```

The next paragraphs describe the usage of all DataReaderView operations. The inherited and abstract operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited and in the data type specific classes in which they are implemented.

Because the DataReaderView closely follows DataReader semantics, a lot of operations are identical. In those cases where the operation on the DataReaderView is identical to the one on the DataReader, no full description is given but the operation on the DataReader or its respective type specific class is referenced.

3.5.9.1 create_querycondition

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public QueryCondition
    create_querycondition
        (int sample_states,
          int view_states,
          int instance_states,
          String query_expression,
          String[] query_parameters);
```

Description

This operation creates a new QueryCondition for the DataReaderView. For a full description please refer to Section 3.5.2.2, *create_querycondition*, on page 371, which describes this operation in detail for the DataReader class.

3.5.9.2 create_readcondition

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public ReadCondition
    create_readcondition
        (int sample_states,
          int view_states,
          int instance_states);
```

Description

This operation creates a new ReadCondition for the DataReaderView. For a full description please refer to Section 3.5.2.3, *create_readcondition*, on page 373, which describes this operation in detail for the DataReader class.

3.5.9.3 delete_contained_entities

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public int
   delete_contained_entities
       (void);
```

Description

This operation deletes all the entitiesthat were created by means of one of the "`create_`" operations on the DataReaderView. For a full description please refer to Section 3.5.2.5, *delete_contained_entities*, on page 375, which describes this operation in detail for the DataReader class.

3.5.9.4 delete_readcondition

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public int
   delete_readcondition
       (ReadCondition a_condition);
```

Description

This operation deletes a ReadCondition or QueryCondition which is attached to the DataReaderView. For a full description please refer to Section 3.5.2.5, *delete_contained_entities*, on page 375, which describes this operation in detail for the DataReader class.

3.5.9.5 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   enable
       (void);
```

3.5.9.6 **get_datareader**

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public DataReader
  get_datareader
      (void);
```

Description

Retrieves the DataReader to which this DataReaderView is attached.

Parameters

```
<none>
```

Return Value

DataReader - A reference to the DataReader.

Detailed Description

This operation returns a reference to the `DataReader` from which the `DataReaderView` was originally created. If the `DataReader` is already deleted, `null` is returned.

3.5.9.7 **get_key_value (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
  get_key_value
      (<data>Holder key_holder,
        long handle);
```

3.5.9.8 **get_qos**

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public int
    get_qos
        (DataReaderViewQosHolder qos);
```

Description

This operation allows access to the existing set of QoS policies for a `DataReaderView`. For a full description please refer to Section 3.5.2.16, *get_qos*, on page 386, which describes this operation in detail for the `DataReader` class.

Parameters

in   DataReaderViewQosHolder   qos - the destination `DataReaderViewQosHolder` object in which the QosPolicy settings will be copied.

Return Value

int - Possible return codes of the operation are:
    `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `DataReaderView` on which this operation is used. This `DataReaderViewQos` is stored at the location pointed to by the `qos` parameter.

Return Code

When the operation returns:

- *RETCODE_OK*   - the existing set of QoSPolicy values applied to this `DataReaderView` has successfully been copied into the specified `DataReaderViewQos` parameter.

- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_ALREADY_DELETED*   - the `DataReaderView` has already been deleted.

- *RETCODE_OUT_OF_RESOURCES*   - the Data Distribution Service ran out of resources to complete this operation.

3.5.9.9 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
    get_status_changes
        (void);
```

3.5.9.10 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

3.5.9.11 lookup_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
long
    lookup_instance
        (<data> instance_data)
```

3.5.9.12 read (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
    read
        (<data>SeqHolder data_values,
          SampleInfoSeqHolder info_seq,
          int max_samples,
          int sample_states,
```

```
                    int view_states,
                    int instance_states);
```

3.5.9.13 **read_instance (abstract)**

This abstract operation is defined as a generic operation, which is implemented by
the `<type>DataReaderView` class. Therefore, to use this operation, the data type
specific implementation of this operation in its respective derived class must be
used. For further explanation see the description for the fictional data type `Foo`
derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   read_instance
      (<data>SeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        int sample_states,
        int view_states,
        int instance_states);
```

3.5.9.14 **read_next_instance (abstract)**

This abstract operation is defined as a generic operation, which is implemented by
the `<type>DataReaderView` class. Therefore, to use this operation, the data type
specific implementation of this operation in its respective derived class must be
used. For further explanation see the description for the fictional data type `Foo`
derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   read_next_instance
      (<data>SeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        int sample_states,
        int view_states,
        int instance_states);
```

3.5.9.15 **read_next_instance_w_condition (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   read_next_instance_w_condition
      (<data>SeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        ReadCondition a_condition);
```

3.5.9.16 **read_next_sample (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   read_next_sample
      (<data>Holder data_values,
        SampleInfoHolder sample_info);
```

3.5.9.17 **read_w_condition (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   read_w_condition
      (<data>SeqHolder data_values,
        SampleInfoSeqHolder info_seq,
```

```
                int max_samples,
                ReadCondition a_condition);
```

3.5.9.18 return_loan (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
    return_loan
        (<data>SeqHolder data_values,
          SampleInfoSeqHolder info_seq);
```

3.5.9.19 set_qos

Scope

```
DDS.DataReaderView
```

Synopsis

```
import DDS.*;
public int
    set_qos
        (DataReaderViewQos qos);
```

Description

This operation replaces the existing set of QosPolicy settings for a `DataReaderView`.

Parameters

in DataReaderViewQos qos - qos contains the new set of QosPolicy settings for the `DataReaderView`.

Return Value

int - Possible return codes of the operation are:
`RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER,`
`RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or`
`RETCODE_IMMUTABLE_POLICY.`

Detailed Description

This operation replaces the existing set of QosPolicy settings for a `DataReaderView`.

The parameter `qos` contains the QosPolicy settings which is checked for self-consistency and mutability. When the application tries to change a QosPolicy setting for an enabled `DataReaderView`, which can only be set before the `DataReaderView` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the presently set QosPolicy settings in case of the immutable QosPolicy settings. Only the mutable QosPolicy settings can be changed.

The set of QosPolicy settings specified by the qos parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided that the operation returned `RETCODE_OK`).

Return Code

When the operation returns:

- *RETCODE_OK* - the new `DataReaderViewQos` is set.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `qos` is not a valid `DataReaderViewQos`. It contains `null` Strings or Strings that do not represent accessible attributes of the datatype.
- *RETCODE_ALREADY_DELETED* - the `DataReaderView` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter `qos` contains an immutable QosPolicy setting with a value different from the one set during enabling of the `DataReaderView`.

3.5.9.20 **take (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
```

```
take
   (<data>SeqHolder data_values,
    SampleInfoSeqHolder info_seq,
    int max_samples,
    int sample_states,
    int view_states,
    int instance_states);
```

3.5.9.21 **take_instance (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   take_instance
      (<data>SeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       long a_handle,
       int sample_states,
       int view_states,
       int instance_states);
```

3.5.9.22 **take_next_instance (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
   take_next_instance
      (<data>SeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       long a_handle,
       int sample_states,
       int view_states,
       int instance_states);
```

3.5.9.23 **take_next_instance_w_condition (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
    take_next_instance_w_condition
        (<data>SeqHolder data_values,
          SampleInfoSeqHolder info_seq,
          int max_samples,
          long a_handle,
          ReadCondition a_condition);
```

3.5.9.24 **take_next_sample (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
    take_next_sample
        (<data>Holder data_values,
          SampleInfoHolder sample_info);
```

3.5.9.25 **take_w_condition (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReaderView` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReaderView` class.

Synopsis

```
import DDS.*;
public int
    take_w_condition
        (<data>SeqHolder data_values,
          SampleInfoSeqHolder info_seq,
```

```
                        int max_samples,
                        ReadCondition a_condition);
```

3.5.10 Class FooDataReaderView

The preprocessor generates from IDL type descriptions the `<type>DataReaderView` classes. For each application data type which is used as Topic data type, a typed class `<type>DataReaderView` is derived from the `DataReaderView` class. In this paragraph, the class `FooDataReaderView` in the namespace `SPACE` describes the operations of these derived `<type>DataReaderView` classes as an example for the fictional type `Foo` (defined in the module `SPACE`).

The interface description of this class is as follows:

```
public interface FooDataReaderView
{
//
// extends interface Entity
//
// public StatusCondition
//    get_statuscondition
//        (void);
// public int
//    get_status_changes
//        (void);
// public int
//    enable
//        (void);
//
// extends interface DataReaderView
//
// public QueryCondition
//    create_querycondition
//        (int sample_states,
//         int view_states,
//         int instance_states,
//         String query_expression,
//         String[] query_parameters);
//
// public ReadCondition
//    create_readcondition
//        (int sample_states,
//         int view_states,
//         int instance_states);
//
// public int
//    delete_contained_entities
//        (void);
//
```

```
                     // public int
                     //    delete_readcondition
                     //        (ReadCondition a_condition);
                     //
                     // public DataReader
                     //    get_datareader
                     //        (void);
                     //
                     // public int
                     //    get_qos
                     //        (DataReaderViewQosHolder qos);
                     //
                     // public int
                     //    set_qos
                     //        (DataReaderViewQos qos);
                     //
                     //
                     // implemented API operations
                     //
                     public int
                        get_key_value
                           (FooHolder key_holder,
                             long handle);

                     public long
                        lookup_instance
                           (Foo instance_data)

                     public int
                        read
                           (FooSeqHolder data_values,
                             SampleInfoSeqHolder info_seq,
                             int max_samples,
                             int sample_states,
                             int view_states,
                             int instance_states);

                     public int
                        read_instance
                           (FooSeqHolder data_values,
                             SampleInfoSeqHolder info_seq,
                             int max_samples,
                             long a_handle,
                             int sample_states,
                             int view_states,
                             int instance_states);

                     public int
                        read_next_instance
                           (FooSeqHolder data_values,
```

```
                    SampleInfoSeqHolder info_seq,
                    int max_samples,
                    long a_handle,
                    int sample_states,
                    int view_states,
                    int instance_states);

        public int
            read_next_instance_w_condition
                (FooSeqHolder data_values,
                    SampleInfoSeqHolder info_seq,
                    int max_samples,
                    long a_handle,
                    ReadCondition a_condition);

        public int
            read_next_sample
                (FooHolder data_values,
                    SampleInfoHolder sample_info);

        public int
            read_w_condition
                (FooSeqHolder data_values,
                    SampleInfoSeqHolder info_seq,
                    int max_samples,
                    ReadCondition a_condition);

        public int
            return_loan
                (FooSeqHolder data_values,
                    SampleInfoSeqHolder info_seq);

        public int
            take
                (FooSeqHolder data_values,
                    SampleInfoSeqHolder info_seq,
                    int max_samples,
                    int sample_states,
                    int view_states,
                    int instance_states);

        public int
            take_instance
                (FooSeqHolder data_values,
                    SampleInfoSeqHolder info_seq,
                    int max_samples,
                    long a_handle,
                    int sample_states,
                    int view_states,
                    int instance_states);
```

```
public int
    take_next_instance
        (FooSeqHolder data_values,
          SampleInfoSeqHolder info_seq,
          int max_samples,
          long a_handle,
          int sample_states,
          int view_states,
          int instance_states);

public int
    take_next_instance_w_condition
        (FooSeqHolder data_values,
          SampleInfoSeqHolder info_seq,
          int max_samples,
          long a_handle,
          ReadCondition a_condition);

public int
    take_next_sample
        (FooHolder data_values,
          SampleInfoHolder sample_info);

public int
    take_w_condition
        (FooSeqHolder data_values,
          SampleInfoSeqHolder info_seq,
          int max_samples,
          ReadCondition a_condition);
```

The next paragraphs describe the usage of all `FooDataReaderView` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.10.1 create_querycondition (inherited)

This operation is inherited and therefore not described here. See the class `DataReaderView` for further explanation.

Synopsis

```
import DDS.*;
public QueryCondition
    create_querycondition
        (int sample_states,
          int view_states,
          int instance_states,
```

```
                          String query_expression,
                          String[] query_parameters);
```

3.5.10.2 **create_readcondition (inherited)**

This operation is inherited and therefore not described here. See the class `DataReaderView` for further explanation.

Synopsis

```
import DDS.*;
public ReadCondition
   create_readcondition
      (int sample_states,
         int view_states,
         int instance_states);
```

3.5.10.3 **delete_contained_entities**

This operation is inherited and therefore not described here. See the class `DataReaderView` for further explanation.

Synopsis

```
import DDS.*;
public int
   delete_contained_entities
      (void);
```

3.5.10.4 **delete_readcondition (inherited)**

This operation is inherited and therefore not described here. See the class `DataReaderView` for further explanation.

Synopsis

```
import DDS.*;
public int
   delete_readcondition
      (ReadCondition a_condition);
```

3.5.10.5 **enable (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   enable
      (void);
```

3.5.10.6 get_datareader (inherited)

This operation is inherited and therefore not described here. See the class `DataReaderView` for further explanation

Synopsis

```
import DDS.*;
public DataReader
   get_datareader
       (void);
```

3.5.10.7 get_key_value

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   get_key_value
       (FooHolder key_holder,
         long handle);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.10.8 get_qos (inherited)

This operation is inherited and therefore not described here. See the class `DataReaderView` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_qos
       (DataReaderViewQosHolder qos);
```

3.5.10.9 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public int
   get_status_changes
       (void);
```

3.5.10.10 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
import DDS.*;
public StatusCondition
   get_statuscondition
      (void);
```

3.5.10.11 **lookup_instance**

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public long
   lookup_instance
      (Foo instance_data)
```

Description

This operation returns the value of the instance handle which corresponds to the `instance_data`. For a full description please refer to Section 3.5.2.67, *lookup_instance*, on page 420, which describes this operation in detail for the `DataReader` class. Note that instances in the `FooDataReaderView` are not defined by the keys of the `TopicDescription` but by the key list in the `DataReaderView` QosPolicy.

3.5.10.12 **read**

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   read
      (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
```

Description

This operation reads a sequence of Foo samples from the FooDataReaderView. For a full description please refer to Section 3.5.2.68, *read*, on page 420, which describes this operation in detail for the DataReader class.

3.5.10.13 read_instance

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
    read_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

Description

This operation reads a sequence of Foo samples of a single instance from the FooDataReaderView. For a full description please refer to Section 3.5.2.69, *read_instance*, on page 424, which describes this operation in detail for the DataReader class.

3.5.10.14 read_next_instance

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
    read_next_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

Description

This operation reads a sequence of Foo samples of the next single instance from the FooDataReaderView. For a full description please refer to Section 3.5.2.70, *read_next_instance*, on page 426, which describes this operation in detail for the DataReader class.

3.5.10.15 read_next_instance_w_condition

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
    read_next_instance_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         ReadCondition a_condition);
```

Description

This operation reads a sequence of Foo samples of the next single instance from the FooDataReaderView, filtered by a ReadCondition or QueryCondition. For a full description please refer to Section 3.5.2.71, *read_next_instance_w_condition*, on page 429, which describes this operation in detail for the DataReader class.

3.5.10.16 read_next_sample

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
    read_next_sample
        (FooHolder data_values,
         SampleInfoHolder sample_info);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.10.17 read_w_condition

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   read_w_condition
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        ReadCondition a_condition);
```

Description

This operation reads a sequence of Foo samples from the FooDataReaderView, filtered by a ReadCondition or QueryCondition. For a full description please refer to Section 3.5.2.73, *read_w_condition*, on page 431, which describes this operation in detail for the DataReader class.

3.5.10.18 **return_loan**

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   return_loan
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq);
```

Description

This operation indicates to the DataReaderView that the application is done accessing the sequence of data_values and info_seq. For a full description please refer to Section 3.5.2.74, *return_loan*, on page 432, which describes this operation in detail for the DataReader class.

3.5.10.19 **set_qos (inherited)**

This operation is inherited and therefore not described here. See the class DataReaderView for further explanation.

Synopsis

```
import DDS.*;
public int
   set_qos
      (DataReaderViewQos qos);
```

3.5.10.20 take

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   take
      (FooSeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       int sample_states,
       int view_states,
       int instance_states);
```

Description

This operation reads a sequence of Foo samples from the FooDataReaderView and by doing so, removes the data from the FooDataReaderView, but not from the FooDataReader that it belongs to. For a full description please refer to Section 3.5.2.77, *take*, on page 435, which describes this operation in detail for the DataReader class.

3.5.10.21 take_instance

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   take_instance
      (FooSeqHolder data_values,
       SampleInfoSeqHolder info_seq,
       int max_samples,
       long a_handle,
       int sample_states,
       int view_states,
       int instance_states);
```

Description

This operation reads a sequence of Foo samples of a single instance from the FooDataReaderView and by doing so, removes the data from the FooDataReaderView, but not from the FooDataReader that it belongs to. For a full description please refer to Section 3.5.2.78, *take_instance*, on page 436, which describes this operation in detail for the DataReader class.

3.5.10.22 take_next_instance

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   take_next_instance
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        int sample_states,
        int view_states,
        int instance_states);
```

Description

This operation reads a sequence of Foo samples of the next single instance from the FooDataReaderView and by doing so, removes the data from the FooDataReaderView, but not from the FooDataReader that it belongs to. For a full description please refer to Section 3.5.2.79, *take_next_instance*, on page 438, which describes this operation in detail for the DataReader class.

3.5.10.23 take_next_instance_w_condition

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   take_next_instance_w_condition
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        long a_handle,
        ReadCondition a_condition);
```

Description

This operation reads a sequence of Foo samples of the next single instance from the FooDataReaderView, filtered by a ReadCondition or QueryCondition and by doing so, removes the data from the FooDataReaderView, but not from the FooDataReader that it belongs to. For a full description please refer to Section 3.5.2.80, *take_next_instance_w_condition*, on page 440, which describes this operation in detail for the DataReader class.

3.5.10.24 take_next_sample

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   take_next_sample
      (FooHolder data_values,
        SampleInfoHolder sample_info);
```

NOTE: This operation is not yet implemented. It is scheduled for a future release.

3.5.10.25 take_w_condition

Scope

```
SPACE.FooDataReaderView
```

Synopsis

```
import DDS.*;
public int
   take_w_condition
      (FooSeqHolder data_values,
        SampleInfoSeqHolder info_seq,
        int max_samples,
        ReadCondition a_condition);
```

Description

This operation reads a sequence of Foo samples from the FooDataReaderView, filtered by a ReadCondition or QueryCondition and by doing so, removes the data from the FooDataReaderView, but not from the FooDataReader that it belongs to. For a full description please refer to Section 3.5.2.82, *take_w_condition*, on page 442, which describes this operation in detail for the DataReader class.

3.6 **QosProvider**

The QosProvider API allows users to specify the QoS settings of their DCPS entities outside of application code in XML. The QosProvider is delivered as part of the DCPS API of Vortex OpenSplice and has no factory. It is not associated with a `DomainParticipant`, so it can be be obtained by a normal allocation.

3.6.1 **Class QosProvider**

The `QosProvider` class provides access to the QoS settings that are specified in an XML file. The interface is as follows:

```
class QosProvider
{
    /* Constructor */
    QosProvider (
        String uri,
        String profile);

    /* API operations */
    int
        get_participant_qos(
            DDS.DomainParticipantQosHolder qos,
            String id);

    int
        get_topic_qos(
            DDS.TopicQosHolder qos,
            String id);

    int
        get_subscriber_qos(
            DDS.SubscriberQosHolder qos,
            String id);

    int
        get_datareader_qos(
            DDS.DataReaderQosHolder qos,
            String id);

    int
        get_publisher_qos(
            DDS.PublisherQosHolder qos,
            String id);

    int
        get_datawriter_qos(
            DDS.DataWriterQosHolder qos,
            String id);
}
```

3.6.1.1 **QosProvider**

Synopsis

```
QosProvider(
    String uri,
    String profile);
```

Description

Constructs a new `QosProvider` based on the provided `uri` and `profile`.

Parameters

in `String uri` - A Uniform Resource Identifier (URI) that points to the location where the QoS profile needs to be loaded from. Currently only URI's with a 'file' scheme that point to an XML file are supported. If profiles and/or QoS settings are not uniquely identifiable by name within the resource pointed to by `uri`, a random one of them will be stored.

in `String profile` - The name of the QoS profile that serves as the default QoS profile for the `get_*_qos(...)` operations.

Return Value

A `QosProvider` instance that is instantiated with all profiles and/or QoS's loaded from the location specified by the provided `uri`.

Initialization of the `QosProvider` will fail under the following conditions:

— No `uri` is provided.
— The resource pointed to by `uri` cannot be found.
— The content of the resource pointed to by `uri` is malformed (*e.g.*, malformed XML).

When initialisation fails (for example, due to a parse error or when the resource identified by `uri` cannot be found), any subsequent operations on the `QosProvider` will return `DDS::RETCODE_PRECONDITION_NOT_MET`.

3.6.1.2 **get_participant_qos**

Synopsis

```
int
    get_participant_qos(
        DDS.DomainParticipantQosHolder qos,
        String id);
```

Description

Resolves the `DDS.DomainParticipantQos` identified by the `id` from the `uri` the `QosProvider` is associated with.

Parameters

inout DDS.DomainParticipantQosHolder qos - The destination `DDS.DomainParticipantQos` in which the QoS policy settings will be copied.

in String id - The fully-qualified name that identifies a QoS within the `uri` associated with the `QosProvider` or a name that identifies a QoS within the `uri` associated with the `QosProvider` instance relative to its default QoS profile. `Id`'s starting with ':: ' are interpreted as fully-qualified names and all others are interpreted as names relative to the default QoS profile of the `QosProvider` instance. When `id` is *NULL* it is interpreted as a non-named QoS within the default QoS profile associated with the `QosProvider`.

Return Value

The operation may return:

- *DDS::RETCODE_OK* - If `qos` has been initialized successfully.
- *DDS::RETCODE_NO_DATA* - If no `DDS.DomainParticipantQos` that matches the provided `id` can be found within the `uri` associated with the `QosProvider`.
- *DDS::RETCODE_BAD_PARAMETER* - If `qos == null`.
- *DDS::RETCODE_PRECONDITION_NOT_MET* - If the `QosProvider` instance is not properly initialized.
- *DDS::RETCODE_OUT_OF_RESOURCES* - If not enough memory is available to perform the operation.
- *DDS::RETCODE_ERROR* - If an internal error occurred.

3.6.1.3 **get_topic_qos**

Synopsis

```
int
    get_topic_qos(
        DDS.TopicQosHolder qos,
        String id);
```

Description

Resolves the `DDS.TopicQos` identified by the `id` from the `uri` the `QosProvider` is associated with.

Parameters

inout DDS.TopicQosHolder qos - The destination `DDS.TopicQos` in which the QoS policy settings will be copied.

in String id - The fully-qualified name that identifies a QoS within the uri associated with the `QosProvider` or a name that identifies a QoS within the `uri` associated with the `QosProvider` instance relative to its default QoS profile. `Id`'s starting with '`::`' are interpreted as fully-qualified names and all others are interpreted as names relative to the default QoS profile of the `QosProvider` instance. When `id` is *NULL* it is interpreted as a non-named QoS within the default QoS profile associated with the `QosProvider`.

Return Value

The operation may return:

- *DDS::RETCODE_OK* - If `qos` has been initialized successfully.
- *DDS::RETCODE_NO_DATA* - If no `DDS.TopicQos` that matches the provided `id` can be found within the `uri` associated with the `QosProvider`.
- *DDS::RETCODE_BAD_PARAMETER* - If `qos == null`.
- *DDS::RETCODE_PRECONDITION_NOT_MET* - If the `QosProvider` instance is not properly initialized.
- *DDS::RETCODE_OUT_OF_RESOURCES* - If not enough memory is available to perform the operation.
- *DDS::RETCODE_ERROR* - If an internal error occurred.

3.6.1.4 get_subscriber_qos

Synopsis

```
int
    get_subscriber_qos(
        DDS.SubscriberQosHolder qos,
        String id);
```

Description

Resolves the `DDS.SubscriberQos` identified by the `id` from the `uri` the `QosProvider` is associated with.

Parameters

inout DDS.SubscriberQosHolder qos - The destination `DDS.SubscriberQos` in which the QoS policy settings will be copied.

in String id - The fully-qualified name that identifies a QoS within the `uri`
associated with the `QosProvider` or a name that identifies a QoS within the
`uri` associated with the `QosProvider` instance relative to its default QoS
profile. `Id`'s starting with ':: ' are interpreted as fully-qualified names and all
others are interpreted as names relative to the default QoS profile of the
`QosProvider` instance. When `id` is *NULL* it is interpreted as a non-named QoS
within the default QoS profile associated with the `QosProvider`.

Return Value

The operation may return:

- *DDS::RETCODE_OK* - If `qos` has been initialized successfully.

- *DDS::RETCODE_NO_DATA* - If no `DDS.SubscriberQos` that matches the
provided `id` can be found within the `uri` associated with the `QosProvider`.

- *DDS::RETCODE_BAD_PARAMETER* - If `qos == null`.

- *DDS::RETCODE_PRECONDITION_NOT_MET* - If the `QosProvider` instance is
not properly initialized.

- *DDS::RETCODE_OUT_OF_RESOURCES* - If not enough memory is available to
perform the operation.

- *DDS::RETCODE_ERROR* - If an internal error occurred.

3.6.1.5 get_datareader_qos

Synopsis

```
int
    get_datareader_qos(
        DDS.DataReaderQosHolder qos,
        String id);
```

Description

Resolves the `DDS.DataReaderQos` identified by the `id` from the `uri` the
`QosProvider` is associated with.

Parameters

inout DDS.DataReaderQosHolder qos - The destination
`DDS.DataReaderQos` in which the QoS policy settings will be copied.

in String id - The fully-qualified name that identifies a QoS within the uri
associated with the `QosProvider` or a name that identifies a QoS within the
`uri` associated with the `QosProvider` instance relative to its default QoS
profile. `Id`'s starting with ':: ' are interpreted as fully-qualified names and all

others are interpreted as names relative to the default QoS profile of the `QosProvider` instance. When `id` is *NULL* it is interpreted as a non-named QoS within the default QoS profile associated with the `QosProvider`.

Return Value

The operation may return:

- *DDS::RETCODE_OK* - If `qos` has been initialized successfully.
- *DDS::RETCODE_NO_DATA* - If no `DDS.DataReaderQos` that matches the provided `id` can be found within the `uri` associated with the `QosProvider`.
- *DDS::RETCODE_BAD_PARAMETER* - If `qos` == null.
- *DDS::RETCODE_PRECONDITION_NOT_MET* - If the `QosProvider` instance is not properly initialized.
- *DDS::RETCODE_OUT_OF_RESOURCES* - If not enough memory is available to perform the operation.
- *DDS::RETCODE_ERROR* - If an internal error occurred.

3.6.1.6 **get_publisher_qos**

Synopsis

```
int
    get_publisher_qos(
        DDS.PublisherQosHolder qos,
        String id);
```

Description

Resolves the `DDS.PublisherQos` identified by the `id` from the `uri` the `QosProvider` is associated with.

Parameters

inout DDS.PublisherQosHolder qos - The destination `DDS.PublisherQos` in which the QoS policy settings will be copied.

in String id - The fully-qualified name that identifies a QoS within the `uri` associated with the `QosProvider` or a name that identifies a QoS within the `uri` associated with the `QosProvider` instance relative to its default QoS profile. `Id`'s starting with ':::' are interpreted as fully-qualified names and all others are interpreted as names relative to the default QoS profile of the `QosProvider` instance. When `id` is *NULL* it is interpreted as a non-named QoS within the default QoS profile associated with the `QosProvider`.

Return Value

The operation may return:

- *DDS::RETCODE_OK* - If qos has been initialized successfully.

- *DDS::RETCODE_NO_DATA* - If no DDS.PublisherQos that matches the provided id can be found within the uri associated with the QosProvider.

- *DDS::RETCODE_BAD_PARAMETER* - If qos == null.

- *DDS::RETCODE_PRECONDITION_NOT_MET* - If the QosProvider instance is not properly initialized.

- *DDS::RETCODE_OUT_OF_RESOURCES* - If not enough memory is available to perform the operation.

- *DDS::RETCODE_ERROR* - If an internal error occurred.

3.6.1.7 **get_datawriter_qos**

Synopsis

```
int
    get_datawriter_qos(
        DDS.DataWriterQosHolder qos,
        String id);
```

Description

Resolves the DDS.DataWriterQos identified by the id from the uri the QosProvider is associated with.

Parameters

inout DDS.DataWriterQosHolder qos - The destination DDS.DataWriterQos in which the QoS policy settings will be copied.

in String id - The fully-qualified name that identifies a QoS within the uri associated with the QosProvider or a name that identifies a QoS within the uri associated with the QosProvider instance relative to its default QoS profile. Id's starting with ':: ' are interpreted as fully-qualified names and all others are interpreted as names relative to the default QoS profile of the QosProvider instance. When id is *NULL* it is interpreted as a non-named QoS within the default QoS profile associated with the QosProvider.

Return Value

The operation may return:

- *DDS::RETCODE_OK* - If qos has been initialized successfully.

- *DDS::RETCODE_NO_DATA* - If no `DDS.DataWriterQos` that matches the provided `id` can be found within the `uri` associated with the `QosProvider`.

- *DDS::RETCODE_BAD_PARAMETER* - If `qos == null`.

- *DDS::RETCODE_PRECONDITION_NOT_MET* - If the `QosProvider` instance is not properly initialized.

- *DDS::RETCODE_OUT_OF_RESOURCES* - If not enough memory is available to perform the operation.

- *DDS::RETCODE_ERROR* - If an internal error occurred.

APPENDICES

A *Quality Of Service*

Each `Entity` is accompanied by an `<Entity>Qos` class that implements the basic mechanism for an application to specify Quality of Service attributes. This class consists of `Entity` specific `QosPolicy` attributes. `QosPolicy` attributes are classes where each class specifies the information that controls an `Entity` related (configurable) attribute of the Data Distribution Service. A `QosPolicy` attribute class is identified as `<name>QosPolicy`.

Affected Entities

Each `Entity` can be configured with a set of `QosPolicy` settings. However, any `Entity` cannot support any `QosPolicy`. For instance, a `DomainParticipant` supports different `QosPolicy` settings than a `Topic` or a `Publisher`. The set of `QosPolicy` settings is implemented as a class of `QosPolicy` classes, identified as `<Entity>Qos`. Each `<Entity>Qos` class only contains those `QosPolicy` classes relevant to the specific `Entity`. The `<Entity>Qos` class serves as the parameter to operations which require a `Qos`. `<Entity>Qos` class is the API implementation of the QoS. Depending on the specific `<Entity>Qos`, it controls the behaviour of a `Topic, DataWriter, DataReader, Publisher, Subscriber, DomainParticipant` or `DomainParticipantFactory`[1].

Basic Usage

The basic way to modify or set the `<Entity>Qos` is by using an `get_qos` operation to get all `QosPolicy` settings from this `Entity` (that is the `<Entity>Qos`), modify several specific `QosPolicy` settings and put them back using an `set_qos` operation to set all `QosPolicy` settings on this `Entity` (that is the `<Entity>Qos`). An example of these operations for the `DataWriterQos` are `get_default_datawriter_qos` and `set_default_datawriter_qos`, which take the `DataWriterQos` as a parameter.

The interface description of this class is as shown below.

```
//public class <name>QosPolicy
// see appendix
//
```

---

1. Note that the `DomainParticipantFactory` is a special kind of entity: it does not inherit from `Entity`, nor does it have a `Listener` or `StatusCondition`, but its behaviour can be controlled by its own set of QosPolicies.

```
//
//public class <Entity>Qos
//
public class DomainParticipantFactoryQos
{
    public EntityFactoryQosPolicy entity_factory;
};

public class DomainParticipantQos
{
    public UserDataQosPolicy user_data;
    public EntityFactoryQosPolicy entity_factory;
    public SchedulingQosPolicy watchdog_scheduling;
    public SchedulingQosPolicy listener_scheduling;
};

public class TopicQos
{
    public TopicDataQosPolicy topic_data;
    public DurabilityQosPolicy durability;
    public DurabilityServiceQosPolicy durability_service;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public OwnershipQosPolicy ownership;
};

public class DataWriterQos
{
    public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public OwnershipStrengthQosPolicy ownership_strength;
    public WriterDataLifecycleQosPolicy
        writer_data_lifecycle;
```

```
};

public class PublisherQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
    public EntityFactoryQosPolicy entity_factory; };

public class DataReaderQos
{
   public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public TimeBasedFilterQosPolicy time_based_filter;
    public ReaderDataLifecycleQosPolicy
       reader_data_lifecycle;
};

public class SubscriberQos
{
   public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
    public EntityFactoryQosPolicy entity_factory; };
//
// implemented API operations
//      <no operations>
```

The following paragraphs describe the usage of each `<Entity>Qos` class.

DataReaderQos

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class DataReaderQos
{
   public DurabilityQosPolicy durability;
   public DeadlineQosPolicy deadline;
```

```
        public LatencyBudgetQosPolicy latency_budget;
        public LivelinessQosPolicy liveliness;
        public ReliabilityQosPolicy reliability;
        public DestinationOrderQosPolicy destination_order;
        public HistoryQosPolicy history;
        public ResourceLimitsQosPolicy resource_limits;
        public UserDataQosPolicy user_data;
        public OwnershipQosPolicy ownership;
        public TimeBasedFilterQosPolicy time_based_filter;
        public ReaderDataLifecycleQosPolicy
            reader_data_lifecycle;
};
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `DataReader`.

Attributes

public DurabilityQosPolicy durability - whether the data should be stored for late joining readers. See Section 3.1.3.3 on page 49 for more detailed information about these settings.

public DeadlineQosPolicy deadline - the period within which a new sample is expected. See Section 3.1.3.1 on page 45 for more detailed information about these settings.

public LatencyBudgetQosPolicy latency_budget - used by the Data Distribution Service for optimization. See Section 3.1.3.8 on page 59 for more detailed information about these settings.

public LivelinessQosPolicy liveliness - the way the liveliness of the `DataReader` is asserted to the Data Distribution Service. See Section 3.1.3.10 on page 62 for more detailed information about these settings.

public ReliabilityQosPolicy reliability - the reliability of the data distribution. See Section 3.1.3.16 on page 79 for more detailed information about these settings.

public DestinationOrderQosPolicy destination_order - the order in which the `DataReader` timely orders the data. See Section 3.1.3.2 on page 47 for more detailed information about these settings.

public HistoryQosPolicy history - how samples should be stored. See Section 3.1.3.7 on page 57 for more detailed information about these settings.

public ResourceLimitsQosPolicy resource_limits - the maximum amount of resources to be used. See Section 3.1.3.17 on page 81 for more detailed information about these settings.

public UserDataQosPolicy user_data - used to attach additional information to the `DataReader`. See Section 3.1.3.22 on page 87 for more detailed information about these settings.

public OwnershipQosPolicy ownership - whether a `DataWriter` exclusively owns an instance. See Section 3.1.3.11 on page 64 for more detailed information about these settings.

public TimeBasedFilterQosPolicy time_based_filter - the maximum data rate at which the `DataReader` will receive changes. See Section 3.1.3.19 on page 84 for more detailed information about these settings.

public ReaderDataLifecycleQosPolicy reader_data_lifecycle - determines whether instance state changes (either `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`) are presented to the user when no corresponding samples are available to communicate them. Also it determines how long an instance state change remains available to a user that does not explicitly consume them. See Section 3.1.3.15 on page 76 for more detailed information about these settings.

Detailed Description

A `QosPolicy` can be set when the `DataReader` is created with the `create_datareader` operation (or modified with the `set_qos` operation). Both operations take the `DataReaderQos` object as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have "immutable" semantics meaning that they can only be specified either at `DataReader` creation time or prior to calling the enable operation on the `DataReader`.

See Section 3.1.3, *Class QosPolicy*, on page 35 for a list of all `<name>QosPolicy` settings, their meaning, characteristics, possible values and applicability to `DataReaders`.

The initial value of the default `DataReaderQos` in the `Subscriber` are given in the following table:

Table 17 DATAREADER_QOS_DEFAULT

QosPolicy	Attribute	Value
durability	kind	VOLATILE_DURABILITY_QOS
	service_cleanup_delay	0
deadline	period	DURATION_INFINITY
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITY
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	false
destination_order	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
user_data	value.length	0
ownership	kind	SHARED_OWNERSHIP_QOS
time_based_filter	minimum_separation	0
reader_data_ lifecycle	autopurge_ nowriter_samples_delay	DURATION_INFINITE
	autopurge_ disposed_samples_delay	DURATION_INFINITE
	autopurge_dispose_all	FALSE
	enable_invalid_samples	TRUE
	invalid_sample_visibility.k ind	MINIMUM_INVALID_SAMPLES

DataWriterQos

Scope

DDS

Synopsis

```
import DDS.*;
```

PRISMTECH

```
public class DataReaderQos
{
    public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public OwnershipStrengthQosPolicy ownership_strength;
    public WriterDataLifecycleQosPolicy
writer_data_lifecycle;
};
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `DataWriter`.

Attributes

public DurabilityQosPolicy durability - whether the data should be stored for late joining readers. See Section 3.1.3.16 on page 79 for more detailed information about these settings.

public DeadlineQosPolicy deadline - the period within which a new sample is written. See Section 3.1.3.1 on page 45 for more detailed information about these settings.

public LatencyBudgetQosPolicy latency_budget - used by the Data Distribution Service for optimization. See Section 3.1.3.8 on page 59 for more detailed information about these settings.

public LivelinessQosPolicy liveliness - the way the liveliness of the `DataWriter` is asserted to the Data Distribution Service. See Section 3.1.3.10 on page 62 for more detailed information about these settings.

public ReliabilityQosPolicy reliability - the reliability of the data distribution. See Section 3.1.3.16 on page 79 for more detailed information about these settings.

public DestinationOrderQosPolicy destination_order - the order in which the `DataReader` timely orders the data. See Section 3.1.3.2 on page 47 for more detailed information about these settings.

public HistoryQosPolicy history - how samples should be stored. See Section 3.1.3.7 on page 57 for more detailed information about these settings.

public ResourceLimitsQosPolicy resource_limits - the maximum amount of resources to be used. See Section 3.1.3.17 on page 81 for more detailed information about these settings.

public TransportPriorityQosPolicy transport_priority - a priority hint for the underlying transport layer. See Section 3.1.3.21 on page 86 for more detailed information about these settings.

public LifespanQosPolicy lifespan - the maximum duration of validity of the data written by the `DataWriter`. See Section 3.1.3.9 on page 61 for more detailed information about these settings.

public UserDataQosPolicy user_data - used to attach additional information to the `DataWriter`. See Section 3.1.3.22 on page 87 for more detailed information about these settings.

public OwnershipQosPolicy ownership - whether a `DataWriter` exclusively owns an instance. See Section 3.1.3.11 on page 64 for more detailed information about these settings.

public OwnershipStrengthQosPolicy ownership_strength - the strength to determine the ownership. See Section 3.1.3.12 on page 67 for more detailed information about these settings.

public WriterDataLifecycleQosPolicy writer_data_lifecycle - whether unregistered instances are disposed of automatically or not. See Section 3.1.3.23 on page 88 for more detailed information about these settings.

Detailed Description

A `QosPolicy` can be set when the `DataWriter` is created with the `create_datawriter` operation (or modified with the `set_qos` operation). Both operations take the `DataWriterQos` object as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have "immutable" semantics meaning that they can only be specified either at `DataWriter` creation time or prior to calling the enable operation on the `DataWriter`.

The initial value of the default `DataWriterQos` in the `Publisher` are given in the following table:

Table 18 DATAWRITER_QOS_DEFAULT

QosPolicy	Attribute	Value
durability	kind	VOLATILE_DURABILITY_QOS
	service_cleanup_delay	0
deadline	period	DURATION_INFINITE
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITY
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	false
destination_order	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
transport_priority	value	0
lifespan	duration	DURATION_INFINITE
user_data	value.length	0
ownership	kind	SHARED_OWNERSHIP_QOS
ownership_strength	value	0
writer_data_lifecycle	autodispose_ unregistered_instances	true

DomainParticipantFactoryQos

Scope

DDS

Synopsis

```
import DDS.*;
public class DomainParticipantFactoryQos
{   public EntityFactoryQosPolicy entity_factory; };
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `DomainParticipantFactory`.

Attributes

EntityFactoryQosPolicy entity_factory - whether a just created `DomainParticipant` should be enabled. See Section 3.1.3.5 on page 55 for more detailed information about these settings.

Detailed Description

The `QosPolicy` cannot be set at creation time, since the `DomainParticipantFactory` is a pre-existing object that can only be obtained with the `DomainParticipantFactory.get_instance` operation or its alias `TheParticipantFactory`. Therefore its `QosPolicy` is initialized to a default value according to the following table:

Table 19 Default Values for DomainParticipantFactoryQos

QosPolicy	Attribute	Value
entity_factory	autoenable_created_entities	TRUE

After creation the `QosPolicy` can be modified with the `set_qos` operation on the `DomainParticipantFactory`, which takes the `DomainParticipantFactoryQos` class as a parameter.

DomainParticipantQos

Scope

DDS

Synopsis

```
import DDS.*;
public class DomainParticipantQos
{   public UserDataQosPolicy user_data;
    public EntityFactoryQosPolicy entity_factory;
    public SchedulingQosPolicy watchdog_scheduling;
    public SchedulingQosPolicy listener_scheduling; };
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `DomainParticipant`.

Attributes

public UserDataQosPolicy user_data - used to attach additional information to the `DomainParticipant`. See Section 3.1.3.22 on page 87 for more detailed information about these settings.

public EntityFactoryQosPolicy entity_factory - whether a just created `Entity` should be enabled. See Section 3.1.3.5 on page 55 for more detailed information about these settings.

public SchedulingQosPolicy watchdog_scheduling - the scheduling parameters used to create the watchdog thread. See Section 3.1.3.18 on page 82 for more detailed information about these settings.

public SchedulingQosPolicy listener_scheduling - the scheduling parameters used to create the listener thread. See Section 3.1.3.18 on page 82 for more detailed information about these settings.

Detailed Description

A `DomainParticipant` will spawn different threads for different purposes:

• A listener thread is spawned to perform the callbacks to all `Listener` objects attached to the various `Entities` contained in the `DomainParticipant`. The scheduling parameters for this thread can be specified in the `listener_scheduling` field of the `DomainParticipantQos`.

• A watchdog thread is spawned to report the the Liveliness of all `Entities` contained in the `DomainParticipant` whose `LivelinessQosPolicyKind` in their `LivelinessQosPolicy` is set to `AUTOMATIC_LIVELINESS_QOS`. The scheduling parameters for this thread can be specified in the `watchdog_scheduling` field of the `DomainParticipantQos`.

A `QosPolicy` can be set when the `DomainParticipant` is created with the `create_participant` operation (or modified with the `set_qos` operation). Both operations take the `DomainParticipantQos` object as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have "immutable" semantics meaning that they can only be specified either at `DomainParticipant` creation time or prior to calling the enable operation on the `DomainParticipant`.

The initial value of the default `DomainParticipantQos` in the `DomainParticipantFactory` are given in the following table:

Table 20 PARTICIPANT_QOS_DEFAULT

QosPolicy	Attribute	Value
user_data	value.length	0
entity_factory	autoenable_created_entities	true
watchdog_scheduling	scheduling_class.kind	SCHEDULE_DEFAULT
	scheduling_priority_kind.kind	PRIORITY_RELATIVE
	scheduling_priority	0
listener_scheduling	scheduling_class.kind	SCHEDULE_DEFAULT
	scheduling_priority_kind.kind	PRIORITY_RELATIVE
	scheduling_priority	0

PublisherQos

Scope

DDS

Synopsis

```
import DDS.*;
public class PublisherQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
    public EntityFactoryQosPolicy entity_factory; };
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `Publisher`.

Attributes

public PresentationQosPolicy presentation - the dependency of changes to data-instances. See Section 3.1.3.14 on page 68 for more detailed information about these settings.

public PartitionQosPolicy partition - the partitions in which the `Publisher` is active. See Section 3.1.3.13 on page 67 for more detailed information about these settings.

public GroupDataQosPolicy group_data - used to attach additional information to the `Publisher`. See Section 3.1.3.6 on page 56 for more detailed information about these settings.

public EntityFactoryQosPolicy entity_factory - whether a just created `DataWriter` should be enabled See Section 3.1.3.5 on page 55 for more detailed information about these settings.

Detailed Description

A `QosPolicy` can be set when the `Publisher` is created with the `create_publisher` operation (or modified with the `set_qos` operation). Both operations take the `PublisherQos` object as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have "immutable" semantics meaning that they can only be specified either at `Publisher` creation time or prior to calling the enable operation on the `Publisher`.

The initial value of the default `PublisherQos` in the `DomainParticipant` are given in the following table:

Table 21 PUBLISHER_QOS_DEFAULT

QosPolicy	Attribute	Value
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	false
	ordered_access	false
partition	name.length	0
group_data	value.length	0
entity_factory	autoenable_created_entities	true

SubscriberQos

Scope

DDS

Synopsis

```
import DDS.*;
public class SubscriberQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
```

```
                    public EntityFactoryQosPolicy entity_factory; };
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `Subscriber`.

Attributes

public PresentationQosPolicy presentation - the dependency of changes to data-instances. See Section 3.1.3.14 on page 68 for more detailed information about these settings.

public PartitionQosPolicy partition - the partitions in which the `Subscriber` is active. See Section 3.1.3.13 on page 67 for more detailed information about these settings.

public GroupDataQosPolicy group_data - used to attach additional information to the `Subscriber`. See Section 3.1.3.6 on page 56 for more detailed information about these settings.

public EntityFactoryQosPolicy entity_factory - whether a just created `DataReader` should be enabled. See Section 3.1.3.5 on page 55 for more detailed information about these settings.

Detailed Description

A `QosPolicy` can be set when the `Subscriber` is created with the `create_subscriber` operation (or modified with the `set_qos` operation). Both operations take the `SubscriberQos` object as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have "immutable" semantics meaning that they can only be specified either at `Subscriber` creation time or prior to calling the enable operation on the `Subscriber`.

The initial value of the default `SubscriberQos` in the `DomainParticipant` are given in the following table:

Table 22 SUBSCRIBER_QOS_DEFAULT

QosPolicy	Attribute	Value
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	false
	ordered_access	false

Table 22 SUBSCRIBER_QOS_DEFAULT

QosPolicy	Attribute	Value
partition	name.length	0
group_data	value.length	0
entity_factory	autoenable_created_entities	true

TopicQos

Scope

```
DDS
```

Synopsis

```
import DDS.*;
public class TopicQos
 {
     public TopicDataQosPolicy topic_data;
     public DurabilityQosPolicy durability;
     public DurabilityServiceQosPolicy durability_service;
     public DeadlineQosPolicy deadline;
     public LatencyBudgetQosPolicy latency_budget;
     public LivelinessQosPolicy liveliness;
     public ReliabilityQosPolicy reliability;
     public DestinationOrderQosPolicy destination_order;
     public HistoryQosPolicy history;
     public ResourceLimitsQosPolicy resource_limits;
     public TransportPriorityQosPolicy transport_priority;
     public LifespanQosPolicy lifespan;
     public OwnershipQosPolicy ownership; };
```

Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `Topic`.

Attributes

public TopicDataQosPolicy topic_data - used to attach additional information to the `Topic`. See Section 3.1.3.20 on page 86 for more detailed information about these settings.

public DurabilityQosPolicy durability - whether the data should be stored for late joining readers. See Section 3.1.3.3 on page 49 for more detailed information about these settings.

public DurabilityServiceQosPolicy durability_service - the behaviour of the "transient/persistent service" of the Data Distribution System regarding Transient and Persistent `Topic` instances. See Section 3.1.3.4 on page 53 for more detailed information about these settings.

public DeadlineQosPolicy deadline - the period within which a new sample is expected or written. See Section 3.1.3.1 on page 45 for more detailed information about these settings.

public LatencyBudgetQosPolicy latency_budget - used by the Data Distribution Service for optimization. See Section 3.1.3.8 on page 59 for more detailed information about these settings.

public LivelinessQosPolicy liveliness - the way the liveliness of the `Topic` is asserted to the Data Distribution Service. See Section 3.1.3.10 on page 62 for more detailed information about these settings.

public ReliabilityQosPolicy reliability - the reliability of the data distribution. See Section 3.1.3.16 on page 79 for more detailed information about these settings.

public DestinationOrderQosPolicy destination_order - the order in which the `DataReader` timely orders the data. See Section 3.1.3.2 on page 47 for more detailed information about these settings.

public HistoryQosPolicy history - how samples should be stored. See Section 3.1.3.7 on page 57 for more detailed information about these settings.

public ResourceLimitsQosPolicy resource_limits - the maximum amount of resources to be used. See Section 3.1.3.17 on page 81 for more detailed information about these settings.

public TransportPriorityQosPolicy transport_priority - a priority hint for the underlying transport layer. See Section 3.1.3.21 on page 86 for more detailed information about these settings.

public LifespanQosPolicy lifespan - the maximum duration of validity of the data written by a `DataWriter`. See Section 3.1.3.9 on page 61 for more detailed information about these settings.

public OwnershipQosPolicy ownership - whether a `DataWriter` exclusively owns an instance. See Section 3.1.3.11 on page 64 for more detailed information about these settings.

Detailed Description

A `QosPolicy` can be set when the `Topic` is created with the `create_topic` operation (or modified with the `set_qos` operation). Both operations take the `TopicQos` object as a parameter. There may be cases where several policies are in

conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have "immutable" semantics meaning that they can only be specified either at `Topic` creation time or prior to calling the enable operation on the `Topic`.

The initial value of the default `TopicQos` in the `DomainParticipant` are given in the following table:

Table 23 TOPIC_QOS_DEFAULT

QosPolicy	Attribute	Value
topic_data	value.length	0
durability	kind	VOLATILE_DURABILITY_QOS
	service_cleanup_delay	0
durability_service	service_cleanup_delay	0
	history_kind	KEEP_LAST_HISTORY_QOS
	history_depth	1
	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
deadline	period	DURATION_INFINITE
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITE
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	false
destination_order	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
transport_priority	value	0
lifespan	duration	DURATION_INFINITE
ownership	kind	SHARED_OWNERSHIP_QOS

B *API Constants and Types*

Duration and Time

```
package DDS;
public final class Duration_t
{
   public int sec = (int)0;
   public int nanosec = (int)0;
   public Duration_t ()
   {
   }
  public Duration_t (int _sec, int _nanosec)
  {
    sec = _sec;
    nanosec = _nanosec;
  }
} // class Duration_t
public interface DURATION_INFINITE_SEC
{
  public static final int value = (int)(0x7fffffff);
}
public interface DURATION_INFINITE_NSEC
{
  public static final int value = (int)(0x7fffffffL);
}
public interface DURATION_ZERO_SEC
{
  public static final int value = (int)(0);
}
public interface DURATION_ZERO_NSEC
{
  public static final int value = (int)(0L);
}
public final class Time_t
{
  public int sec = (int)0;
  public int nanosec = (int)0;
  public Time_t ()
  {
  }
  public Time_t (int _sec, int _nanosec)
  {
    sec = _sec;
```

```
      nanosec = _nanosec;
   }
} // class Time_t
```

Pre-defined Values

```java
public interface HANDLE_NIL
{
    public static final long value = (long)(0L);
}
public interface LENGTH_UNLIMITED
{
  public static final int value = (int)((int)-1);
}
public interface TIMESTAMP_INVALID_SEC
{
  public static final int value = (int)((int)-1);
}
public interface TIMESTAMP_INVALID_NSEC
{
  public static final int value = (int)(0xffffffffL);
}
```

Return Codes

```java
public interface RETCODE_OK
{
  public static final int value = (int)(0);
}
public interface RETCODE_ERROR
{
  public static final int value = (int)(1);
}
public interface RETCODE_UNSUPPORTED
{
  public static final int value = (int)(2);
}
public interface RETCODE_BAD_PARAMETER
{
  public static final int value = (int)(3);
}
public interface RETCODE_PRECONDITION_NOT_MET
{
  public static final int value = (int)(4);
}
public interface RETCODE_OUT_OF_RESOURCES
{
  public static final int value = (int)(5);
}
public interface RETCODE_NOT_ENABLED
{
```

```
            public static final int value = (int)(6);
          }
          public interface RETCODE_IMMUTABLE_POLICY
          {
            public static final int value = (int)(7);
          }
          public interface RETCODE_INCONSISTENT_POLICY
          {
            public static final int value = (int)(8);
          }
          public interface RETCODE_ALREADY_DELETED
          {
            public static final int value = (int)(9);
          }
          public interface RETCODE_TIMEOUT
          {
            public static final int value = (int)(10);
          }
          public interface RETCODE_NO_DATA
          {
            public static final int value = (int)(11);
          }
          public interface RETCODE_ILLEGAL_OPERATION
          {
            public static final int value = (int)(12);
          }
```

Status to Support Listeners and Conditions

```
          public interface INCONSISTENT_TOPIC_STATUS
          {
            public static final int value = (int)((int)(0x0001L << 0L));
          }
          public interface OFFERED_DEADLINE_MISSED_STATUS
          {
            public static final int value = (int)((int)(0x0001L << 1L));
          }
          public interface REQUESTED_DEADLINE_MISSED_STATUS
          {
            public static final int value = (int)((int)(0x0001L << 2L));
          }
          public interface OFFERED_INCOMPATIBLE_QOS_STATUS
          {
            public static final int value = (int)((int)(0x0001L << 5L));
          }
          public interface REQUESTED_INCOMPATIBLE_QOS_STATUS
          {
            public static final int value = (int)((int)(0x0001L << 6L));
          }
          public interface SAMPLE_LOST_STATUS
```

```
                {
                  public static final int value = (int)((int)(0x0001L << 7L));
                }
                public interface SAMPLE_REJECTED_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 8L));
                }
                public interface DATA_ON_READERS_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 9L));
                }
                public interface DATA_AVAILABLE_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 10L));
                }
                public interface LIVELINESS_LOST_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 11L));
                }
                public interface LIVELINESS_CHANGED_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 12L));
                }
                public interface PUBLICATION_MATCH_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 13L));
                }
                public interface SUBSCRIPTION_MATCH_STATUS
                {
                  public static final int value = (int)((int)(0x0001L << 14L));
                }
                public interface ALL_DATA_DISPOSED_TOPIC_STATUS
                {
                    public static final int value = (int)((int)(0x0001L << 31L));
                }
                /* Note: ANY_STATUS is deprecated, please use spec version
                 * specific constants.  e.g. STATUS_MASK_ANY_V1_2
                 */
                public interface ANY_STATUS
                {
                  public static final int value = (int)(0x7FE7L);
                }

                public interface STATUS_MASK_ANY_V1_2
                {
                  public static final int value = (int)(0x7FE7L);
                }

                public interface STATUS_MASK_NONE
                {
```

```
              public static final int value = (int)(0x0L);
          }
```

States

Sample States to Support Reads

```
public interface READ_SAMPLE_STATE
{
  public static final int value = (int)((int)(0x0001L << 0L));
}
public interface NOT_READ_SAMPLE_STATE
{
  public static final int value = (int)((int)(0x0001L << 1L));
}
// This is a bit mask SampleStateKind
public interface ANY_SAMPLE_STATE
{
  public static final int value = (int)(0xffffL);
}
```

View States to Support Reads

```
public interface NEW_VIEW_STATE
{
  public static final int value = (int)((int)(0x0001L << 0L));
}
public interface NOT_NEW_VIEW_STATE
{
  public static final int value = (int)((int)(0x0001L << 1L));
}
// This is a bit mask ViewStateKind
public interface ANY_VIEW_STATE
{
  public static final int value = (int)(0xffffL);
}
```

Instance States to Support Reads

```
public interface ALIVE_INSTANCE_STATE
{
  public static final int value = (int)((int)(0x0001L << 0L));
}
public interface NOT_ALIVE_DISPOSED_INSTANCE_STATE
{
  public static final int value = (int)((int)(0x0001L << 1L));
}
public interface NOT_ALIVE_NO_WRITERS_INSTANCE_STATE
{
  public static final int value = (int)((int)(0x0001L << 2L));
}
```

```
                    // This is a bit mask InstanceStateKind
                    public interface ANY_INSTANCE_STATE
                    {
                      public static final int value = (int)(0xffffL);
                    }
                    public interface NOT_ALIVE_INSTANCE_STATE
                    {
                      public static final int value = (int)(0x006L);
                    }
```

QosPolicy

Names

```
                    public interface USERDATA_QOS_POLICY_NAME
                    {
                      public static final String value = "UserData";
                    }
                    public interface DURABILITY_QOS_POLICY_NAME
                    {
                      public static final String value = "Durability";
                    }
                    public interface PRESENTATION_QOS_POLICY_NAME
                    {
                      public static final String value = "Presentation";
                    }
                    public interface DEADLINE_QOS_POLICY_NAME
                    {
                      public static final String value = "Deadline";
                    }
                    public interface LATENCYBUDGET_QOS_POLICY_NAME
                    {
                      public static final String value = "LatencyBudget";
                    }
                    public interface OWNERSHIP_QOS_POLICY_NAME
                    {
                      public static final String value = "Ownership";
                    }
                    public interface OWNERSHIPSTRENGTH_QOS_POLICY_NAME
                    {
                      public static final String value = "OwnershipStrength";
                    }
                    public interface LIVELINESS_QOS_POLICY_NAME
                    {
                      public static final String value = "Liveliness";
                    }
                    public interface TIMEBASEDFILTER_QOS_POLICY_NAME
                    {
                      public static final String value = "TimeBasedFilter";
                    }
```

```java
public interface PARTITION_QOS_POLICY_NAME
{
  public static final String value = "Partition";
}
public interface RELIABILITY_QOS_POLICY_NAME
{
  public static final String value = "Reliability";
}
public interface DESTINATIONORDER_QOS_POLICY_NAME
{
  public static final String value = "DestinationOrder";
}
public interface HISTORY_QOS_POLICY_NAME
{
  public static final String value = "History";
}
public interface RESOURCELIMITS_QOS_POLICY_NAME
{
  public static final String value = "ResourceLimits";
}
public interface ENTITYFACTORY_QOS_POLICY_NAME
{
  public static final String value = "EntityFactory";
}
public interface WRITERDATALIFECYCLE_QOS_POLICY_NAME
{
  public static final String value =
    "WriterDataLifecycle";
}
public interface READERDATALIFECYCLE_QOS_POLICY_NAME
{
  public static final String value = "ReaderDataLifecycle";
}
public interface TOPICDATA_QOS_POLICY_NAME
{
  public static final String value = "TopicData";
}
public interface GROUPDATA_QOS_POLICY_NAME
{
  public static final String value = "GroupData";
}
public interface TRANSPORTPRIORITY_QOS_POLICY_NAME
{
  public static final String value = "TransportPriority";
}
public interface LIFESPAN_QOS_POLICY_NAME
{
  public static final String value = "Lifespan";
}
public interface DURABILITYSERVICE_QOS_POLICY_NAME
```

```
{
  public static final String value = "DurabilityService";
}
```

Identifications

```
public interface INVALID_QOS_POLICY_ID
{
  public static final int value = (int)(0);
}
public interface USERDATA_QOS_POLICY_ID
{
  public static final int value = (int)(1);
}
public interface DURABILITY_QOS_POLICY_ID
{
  public static final int value = (int)(2);
}
public interface PRESENTATION_QOS_POLICY_ID
{
  public static final int value = (int)(3);
}
public interface DEADLINE_QOS_POLICY_ID
{
  public static final int value = (int)(4);
}
public interface LATENCYBUDGET_QOS_POLICY_ID
{
  public static final int value = (int)(5);
}
public interface OWNERSHIP_QOS_POLICY_ID
{
  public static final int value = (int)(6);
}
public interface OWNERSHIPSTRENGTH_QOS_POLICY_ID
{
  public static final int value = (int)(7);
}
public interface LIVELINESS_QOS_POLICY_ID
{
  public static final int value = (int)(8);
}
public interface TIMEBASEDFILTER_QOS_POLICY_ID
{
  public static final int value = (int)(9);
}
public interface PARTITION_QOS_POLICY_ID
{
  public static final int value = (int)(10);
}
```

```
public interface RELIABILITY_QOS_POLICY_ID
{
  public static final int value = (int)(11);
}
public interface DESTINATIONORDER_QOS_POLICY_ID
{
  public static final int value = (int)(12);
}
public interface HISTORY_QOS_POLICY_ID
{
  public static final int value = (int)(13);
}
public interface RESOURCELIMITS_QOS_POLICY_ID
{
  public static final int value = (int)(14);
}
public interface ENTITYFACTORY_QOS_POLICY_ID
{
  public static final int value = (int)(15);
}
public interface WRITERDATALIFECYCLE_QOS_POLICY_ID
{
  public static final int value = (int)(16);
}
public interface READERDATALIFECYCLE_QOS_POLICY_ID
{
  public static final int value = (int)(17);
}
public interface TOPICDATA_QOS_POLICY_ID
{
  public static final int value = (int)(18);
}
public interface GROUPDATA_QOS_POLICY_ID
{
  public static final int value = (int)(19);
}
public interface TRANSPORTPRIORITY_QOS_POLICY_ID
{
  public static final int value = (int)(20);
}
public interface LIFESPAN_QOS_POLICY_ID
{
  public static final int value = (int)(21);
}
public interface DURABILITYSERVICE_QOS_POLICY_ID
{
  public static final int value = (int)(22);
}
```

Appendices

C *Platform Specific Model IDL Interface*

dds_dcps.idl

```
#define DOMAINID_TYPE_NATIVE long
#define HANDLE_TYPE_NATIVE long long
#define HANDLE_NIL_NATIVE 0
#define BUILTIN_TOPIC_KEY_TYPE_NATIVE long
#define TheParticipantFactory
#define PARTICIPANT_QOS_DEFAULT
#define TOPIC_QOS_DEFAULT
#define PUBLISHER_QOS_DEFAULT
#define SUBSCRIBER_QOS_DEFAULT
#define DATAWRITER_QOS_DEFAULT
#define DATAREADER_QOS_DEFAULT
#define DATAWRITER_QOS_USE_TOPIC_QOS
#define DATAREADER_QOS_USE_TOPIC_QOS

module DDS {
    typedef DOMAINID_TYPE_NATIVE DomainId_t;
    typedef HANDLE_TYPE_NATIVE InstanceHandle_t;
    typedef BUILTIN_TOPIC_KEY_TYPE_NATIVE
       BuiltinTopicKey_t[3];
    typedef sequence<InstanceHandle_t> InstanceHandleSeq;
    typedef long ReturnCode_t;
    typedef long QosPolicyId_t;
    typedef sequence<string> StringSeq;
    struct Duration_t {
    long sec;
    unsigned long nanosec;
    };
    struct Time_t {
       long sec;
       unsigned long nanosec;
    };
```

Pre-defined Values

```
    const InstanceHandle_t HANDLE_NIL= HANDLE_NIL_NATIVE;
   const long LENGTH_UNLIMITED= -1;
   const long DURATION_INFINITE_SEC= 0x7fffffff;
   const unsigned long DURATION_INFINITE_NSEC= 0x7fffffff;
```

```
                   const long DURATION_ZERO_SEC= 0;
                   const unsigned long DURATION_ZERO_NSEC= 0;
                   const long TIMESTAMP_INVALID_SEC= -1;
                   const unsigned long TIMESTAMP_INVALID_NSEC= 0xffffffff;
                   const DomainId_t DOMAIN_ID_DEFAULT= 0x7fffffff;
```

Return Codes

```
         const ReturnCode_t RETCODE_OK                   = 0;
         const ReturnCode_t RETCODE_ERROR                = 1;
         const ReturnCode_t RETCODE_UNSUPPORTED          = 2;
         const ReturnCode_t RETCODE_BAD_PARAMETER        = 3;
         const ReturnCode_t RETCODE_PRECONDITION_NOT_MET = 4;
         const ReturnCode_t RETCODE_OUT_OF_RESOURCES     = 5;
         const ReturnCode_t RETCODE_NOT_ENABLED          = 6;
         const ReturnCode_t RETCODE_IMMUTABLE_POLICY     = 7;
         const ReturnCode_t RETCODE_INCONSISTENT_POLICY  = 8;
         const ReturnCode_t RETCODE_ALREADY_DELETED      = 9;
         const ReturnCode_t RETCODE_TIMEOUT              = 10;
         const ReturnCode_t RETCODE_NO_DATA              = 11;
         const ReturnCode_t RETCODE_ILLEGAL_OPERATION    = 12;
```

Status to Support Listeners and Conditions

```
         typedef unsigned long StatusKind;
         typedef unsigned long StatusMask;
         // bit mask StatusKind
         const StatusKind INCONSISTENT_TOPIC_STATUS        = 0x0001 << 0;
         const StatusKind OFFERED_DEADLINE_MISSED_STATUS   = 0x0001 << 1;
         const StatusKind REQUESTED_DEADLINE_MISSED_STATUS = 0x0001 << 2;
         const StatusKind OFFERED_INCOMPATIBLE_QOS_STATUS  = 0x0001 << 5;
         const StatusKind REQUESTED_INCOMPATIBLE_QOS_STATUS= 0x0001 << 6;
         const StatusKind SAMPLE_LOST_STATUS               = 0x0001 << 7;
         const StatusKind SAMPLE_REJECTED_STATUS           = 0x0001 << 8;
         const StatusKind DATA_ON_READERS_STATUS           = 0x0001 << 9;
         const StatusKind DATA_AVAILABLE_STATUS            = 0x0001 << 10;
         const StatusKind LIVELINESS_LOST_STATUS           = 0x0001 << 11;
         const StatusKind LIVELINESS_CHANGED_STATUS        = 0x0001 << 12;
         const StatusKind PUBLICATION_MATCHED_STATUS       = 0x0001 << 13;
         const StatusKind SUBSCRIPTION_MATCHED_STATUS      = 0x0001 << 14;
         const StatusKind ALL_DATA_DISPOSED_TOPIC_STATUS   = 0x0001 << 31;

         struct InconsistentTopicStatus {
            long total_count;
            long total_count_change;
         };

         struct SampleLostStatus {
            long total_count;
            long total_count_change;
         };
```

```
enum SampleRejectedStatusKind {
   NOT_REJECTED,
   REJECTED_BY_INSTANCES_LIMIT,
   REJECTED_BY_SAMPLES_LIMIT,
   REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
};

struct SampleRejectedStatus {
   long total_count;
   long total_count_change;
   SampleRejectedStatusKind last_reason;
   InstanceHandle_t last_instance_handle;
};

struct LivelinessChangedStatus {
   long alive_count;
   long not_alive_count;
   long alive_count_change;
   long not_alive_count_change;
   InstanceHandle_t last_publication_handle;
};

struct LivelinessChangedStatus {
   long active_count;
   long inactive_count;
   long active_count_change;
   long inactive_count_change;
};

struct OfferedDeadlineMissedStatus {
   long total_count;
   long total_count_change;
   InstanceHandle_t last_instance_handle;
};

struct RequestedDeadlineMissedStatus {
   long total_count;
   long total_count_change;
   InstanceHandle_t last_instance_handle;
};

struct QosPolicyCount {
   QosPolicyId_t policy_id;
   long count;
};

typedef sequence<QosPolicyCount> QosPolicyCountSeq;

struct OfferedIncompatibleQosStatus {
```

```
              long total_count;
              long total_count_change;
              QosPolicyId_t last_policy_id;
              QosPolicyCountSeq policies;
           };

           struct RequestedIncompatibleQosStatus {
              long total_count;
              long total_count_change;
              QosPolicyId_t last_policy_id;
              QosPolicyCountSeq policies;
           };

           struct PublicationMatchedStatus {
              long total_count;
              long total_count_change;
              long current_count;
              long current_count_change;
              InstanceHandle_t last_subscription_handle;

           struct SubscriptionMatchedStatus {
              long total_count;
              long total_count_change;
              long current_count;
              long current_count_change;
              InstanceHandle_t last_publication_handle;
           };

           struct AllDataDisposedTopicStatus {
              long total_count;
              long total_count_change;
           };
```

Listeners

```
        interface Listener;
        interface Entity;
        interface TopicDescription;
        interface Topic;
        interface ContentFilteredTopic;
        interface MultiTopic;
        interface DataWriter;
        interface DataReader;
        interface Subscriber;
        interface Publisher;
        typedef sequence<Topic> TopicSeq;

        typedef sequence<DataReader> DataReaderSeq;
        interface Listener {
        };
```

```
interface TopicListener : Listener {
void
on_inconsistent_topic(
    in Topic the_topic,
    in InconsistentTopicStatus status);
};

interface ExtTopicListener : TopicListener {
void
on_all_data_disposed(in Topic the_topic);
};

interface DataWriterListener : Listener {

void
on_offered_deadline_missed(
    in DataWriter writer,
    in OfferedDeadlineMissedStatus status);

void
on_offered_incompatible_qos(
    in DataWriter writer,
    in OfferedIncompatibleQosStatus status);

void
on_liveliness_lost(
    in DataWriter writer,
    in LivelinessLostStatus status);

void
on_publication_matched(
    in DataWriter writer,
    in PublicationMatchedStatus status);
};

interface PublisherListener : DataWriterListener {
};

interface DataReaderListener : Listener {

void
on_requested_deadline_missed(
    in DataReader reader,
    in RequestedDeadlineMissedStatus status);


void
on_requested_incompatible_qos(
```

```
        in DataReader reader,
        in RequestedIncompatibleQosStatus status);

    void
    on_sample_rejected(
        in DataReader reader,
        in SampleRejectedStatus status);

    void
    on_liveliness_changed(
        in DataReader reader,
        in LivelinessChangedStatus status);

    void
    on_data_available(
        in DataReader reader);

    void
    on_subscription_matched(
        in DataReader reader,
        in SubscriptionMatchedStatus status);

    void
    on_sample_lost(
        in DataReader reader,
        in SampleLostStatus status);
    };

  interface SubscriberListener : DataReaderListener {

    void
    on_data_on_readers(
        in Subscriber subs);
    };

  interface DomainParticipantListener : TopicListener,
                        PublisherListener,
                        SubscriberListener {
    };

 interface ExtDomainParticipantListener : DomainParticipantListener,
                        ExtTopicListener {
    };
```

Conditions

```
    interface Condition {

    boolean
```

```
get_trigger_value();
};

typedef sequence<Condition> ConditionSeq;
interface WaitSet {

ReturnCode_t
wait(
    inout ConditionSeq active_conditions,
    in Duration_t timeout);

ReturnCode_t
attach_condition(
    in Condition cond);

ReturnCode_t
detach_condition(
    in Condition cond);

ReturnCode_t
get_conditions(
    inout ConditionSeq attached_conditions);
};

interface GuardCondition : Condition {

ReturnCode_t
set_trigger_value(
    in boolean value);
};

interface StatusCondition : Condition {

StatusMask
get_enabled_statuses();

ReturnCode_t
set_enabled_statuses(
    in StatusMask mask);
Entity
get_entity();
};

// Sample states to support reads
typedef unsigned long SampleStateKind;
typedef sequence <SampleStateKind> SampleStateSeq;
const SampleStateKind READ_SAMPLE_STATE= 0x0001 << 0;
const SampleStateKind NOT_READ_SAMPLE_STATE= 0x0001 << 1;

// This is a bit mask SampleStateKind
```

```
typedef unsigned long SampleStateMask;
const SampleStateMask ANY_SAMPLE_STATE= 0xffff;

// View states to support reads
typedef unsigned long ViewStateKind;
typedef sequence<ViewStateKind> ViewStateSeq;
const ViewStateKind NEW_VIEW_STATE= 0x0001 << 0;
const ViewStateKind NOT_NEW_VIEW_STATE= 0x0001 << 1;

// This is a bit mask ViewStateKind
typedef unsigned long ViewStateMask;
const ViewStateMask ANY_VIEW_STATE= 0xffff;

// Instance states to support reads
typedef unsigned long InstanceStateKind;
typedef sequence<InstanceStateKind> InstanceStateSeq;
const InstanceStateKind ALIVE_INSTANCE_STATE = 0x0001
    << 0;
const InstanceStateKind NOT_ALIVE_DISPOSED_INSTANCE_STATE
  = 0x0001 << 1;
const InstanceStateKind
   'NOT_ALIVE_NO_WRITERS_INSTANCE_STATE = 0x0001 << 2;

// This is a bit mask InstanceStateKind
typedef unsigned long InstanceStateMask;
const InstanceStateMask ANY_INSTANCE_STATE= 0xffff;
const InstanceStateMask NOT_ALIVE_INSTANCE_STATE= 0x006;

interface ReadCondition : Condition {
SampleStateMask
get_sample_state_mask();

ViewStateMask
get_view_state_mask();

InstanceStateMask
get_instance_state_mask();

DataReader
get_datareader();
 };

 interface QueryCondition : ReadCondition {
string
get_query_expression();

ReturnCode_t
get_query_parameters(
    inout StringSeq query_parameters);
```

```
        ReturnCode_t
        set_query_parameters(
            in StringSeq query_parameters);
        };
```

QoS

```
    const string USERDATA_QOS_POLICY_NAME= "UserData";
    const string DURABILITY_QOS_POLICY_NAME= "Durability";
    const string PRESENTATION_QOS_POLICY_NAME =
        "Presentation";
    const string DEADLINE_QOS_POLICY_NAME= "Deadline";
    const string LATENCYBUDGET_QOS_POLICY_NAME =
        "LatencyBudget";
    const string OWNERSHIP_QOS_POLICY_NAME= "Ownership";
    const string OWNERSHIPSTRENGTH_QOS_POLICY_NAME=
        "OwnershipStrength";
    const string LIVELINESS_QOS_POLICY_NAME= "Liveliness";
    const string TIMEBASEDFILTER_QOS_POLICY_NAME=
        "TimeBasedFilter";
    const string PARTITION_QOS_POLICY_NAME= "Partition";
    const string RELIABILITY_QOS_POLICY_NAME= "Reliability";
    const string DESTINATIONORDER_QOS_POLICY_NAME=
        "DestinationOrder";
    const string HISTORY_QOS_POLICY_NAME= "History";
    const string RESOURCELIMITS_QOS_POLICY_NAME=
        "ResourceLimits";
    const string ENTITYFACTORY_QOS_POLICY_NAME=
        ' "EntityFactory";
    const string WRITERDATALIFECYCLE_QOS_POLICY_NAME=
        "WriterDataLifecycle";
    const string READERDATALIFECYCLE_QOS_POLICY_NAME=
        "ReaderDataLifecycle";
    const string TOPICDATA_QOS_POLICY_NAME= "TopicData";
    const string GROUPDATA_QOS_POLICY_NAME= "GroupData";
    const string TRANSPORTPRIORITY_QOS_POLICY_NAME=
        "TransportPriority";
    const string LIFESPAN_QOS_POLICY_NAME= "Lifespan";
    const string DURABILITYSERVICE_QOS_POLICY_NAME=
        "DurabilityService";
    const QosPolicyId_t INVALID_QOS_POLICY_ID           = 0;
    const QosPolicyId_t USERDATA_QOS_POLICY_ID          = 1;
    const QosPolicyId_t DURABILITY_QOS_POLICY_ID        = 2;
    const QosPolicyId_t PRESENTATION_QOS_POLICY_ID      = 3;
    const QosPolicyId_t DEADLINE_QOS_POLICY_ID          = 4;
    const QosPolicyId_t LATENCYBUDGET_QOS_POLICY_ID     = 5;
    const QosPolicyId_t OWNERSHIP_QOS_POLICY_ID         = 6;
    const QosPolicyId_t OWNERSHIPSTRENGTH_QOS_POLICY_ID = 7;
    const QosPolicyId_t LIVELINESS_QOS_POLICY_ID        = 8;
    const QosPolicyId_t TIMEBASEDFILTER_QOS_POLICY_ID   = 9;
```

```
         const QosPolicyId_t PARTITION_QOS_POLICY_ID          = 10;
         const QosPolicyId_t RELIABILITY_QOS_POLICY_ID        = 11;
         const QosPolicyId_t DESTINATIONORDER_QOS_POLICY_ID   = 12;
         const QosPolicyId_t HISTORY_QOS_POLICY_ID            = 13;
         const QosPolicyId_t RESOURCELIMITS_QOS_POLICY_ID     = 14;
         const QosPolicyId_t ENTITYFACTORY_QOS_POLICY_ID      = 15;
         const QosPolicyId_t WRITERDATALIFECYCLE_QOS_POLICY_ID= 16;
         const QosPolicyId_t READERDATALIFECYCLE_QOS_POLICY_ID= 17;
         const QosPolicyId_t TOPICDATA_QOS_POLICY_ID          = 18;
         const QosPolicyId_t GROUPDATA_QOS_POLICY_ID          = 19;
         const QosPolicyId_t TRANSPORTPRIORITY_QOS_POLICY_ID  = 20;
         const QosPolicyId_t LIFESPAN_QOS_POLICY_ID           = 21;
         const QosPolicyId_t DURABILITYSERVICE_QOS_POLICY_ID  = 22;

     struct UserDataQosPolicy {
         sequence<octet> value;
     };

     struct TopicDataQosPolicy {
         sequence<octet> value;
     };

     struct GroupDataQosPolicy {
         sequence<octet> value;
     };

     struct TransportPriorityQosPolicy {
         long value;
     };

     struct LifespanQosPolicy {
         Duration_t duration;
     };

     enum DurabilityQosPolicyKind {
         VOLATILE_DURABILITY_QOS,
         TRANSIENT_LOCAL_DURABILITY_QOS,
         TRANSIENT_DURABILITY_QOS,
         PERSISTENT_DURABILITY_QOS
     };

     struct DurabilityQosPolicy {
         DurabilityQosPolicyKind kind;
     };

     enum PresentationQosPolicyAccessScopeKind {
         INSTANCE_PRESENTATION_QOS,
         TOPIC_PRESENTATION_QOS,
         GROUP_PRESENTATION_QOS
     };
```

```
struct PresentationQosPolicy {
   PresentationQosPolicyAccessScopeKind access_scope;
   boolean coherent_access;
   boolean ordered_access;
};

struct DeadlineQosPolicy {
   Duration_t period;
};

struct LatencyBudgetQosPolicy {
   Duration_t duration;
};

enum OwnershipQosPolicyKind {
   SHARED_OWNERSHIP_QOS,
   EXCLUSIVE_OWNERSHIP_QOS
};

struct OwnershipQosPolicy {
   OwnershipQosPolicyKind kind;
};

struct OwnershipStrengthQosPolicy {
   long value;
};

enum LivelinessQosPolicyKind {
   AUTOMATIC_LIVELINESS_QOS,
   MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
   MANUAL_BY_TOPIC_LIVELINESS_QOS
};

struct LivelinessQosPolicy {
   LivelinessQosPolicyKind kind;
   Duration_t lease_duration;
};

struct TimeBasedFilterQosPolicy {
   Duration_t minimum_separation;
};

struct PartitionQosPolicy {
StringSeq name;
};

enum ReliabilityQosPolicyKind {
   BEST_EFFORT_RELIABILITY_QOS,
   RELIABLE_RELIABILITY_QOS
```

```
      };

      struct ReliabilityQosPolicy {
         ReliabilityQosPolicyKind kind;
         Duration_t max_blocking_time;
         boolean synchronous;
      };

      enum DestinationOrderQosPolicyKind {
         BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
         BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
      };

      struct DestinationOrderQosPolicy {
         DestinationOrderQosPolicyKind kind;
      };

      enum HistoryQosPolicyKind {
         KEEP_LAST_HISTORY_QOS,
         KEEP_ALL_HISTORY_QOS
      };

      struct HistoryQosPolicy {
         HistoryQosPolicyKind kind;
         long depth;
      };

      struct ResourceLimitsQosPolicy {
         long max_samples;
         long max_instances;
         long max_samples_per_instance;
      };

      struct EntityFactoryQosPolicy {
         boolean autoenable_created_entities;
      };

      struct WriterDataLifecycleQosPolicy {
         boolean autodispose_unregistered_instances;
      };

   public class InvalidSampleVisibilityQosPolicyKind
   { public static final InvalidSampleVisibilityQosPolicyKind
      NO_INVALID_SAMPLES;
     public static final InvalidSampleVisibilityQosPolicyKind
      MINIMUM_INVALID_SAMPLES;
     public static final InvalidSampleVisibilityQosPolicyKind
      ALL_INVALID_SAMPLES; };

   public final class InvalidSampleVisibilityQosPolicy
```

PRISMTECH

```
                    { public InvalidSampleVisibilityQosPolicyKind kind; };

               public final class ReaderDataLifecycleQosPolicy
               { public Duration_t autopurge_nowriter_samples_delay;
                 public Duration_t autopurge_disposed_samples_delay;
                 public boolean autopurge_dispose_all;
                 public boolean enable_invalid_samples;
                 public InvalidSampleVisibilityQosPolicy
                    invalid_sample_visibility; };

                struct DurabilityServiceQosPolicy {
                    Duration_t service_cleanup_delay;
                    HistoryQosPolicyKind history_kind;
                    long history_depth;
                    long max_samples;
                    long max_instances;
                    long max_samples_per_instance;
                };

                struct DomainParticipantFactoryQos {
                    EntityFactoryQosPolicy entity_factory;
                };

                struct DomainParticipantQos {
                   UserDataQosPolicy user_data;
                   EntityFactoryQosPolicy entity_factory;
                };

                struct TopicQos {
                   TopicDataQosPolicy topic_data;
                   DurabilityQosPolicy durability;
                   DurabilityServiceQosPolicy durability_service;
                   DeadlineQosPolicy deadline;
                   LatencyBudgetQosPolicy latency_budget;
                   LivelinessQosPolicy liveliness;
                   ReliabilityQosPolicy reliability;
                   DestinationOrderQosPolicy destination_order;
                   HistoryQosPolicy history;
                   ResourceLimitsQosPolicy resource_limits;
                   TransportPriorityQosPolicy transport_priority;
                   LifespanQosPolicy lifespan;
                   OwnershipQosPolicy ownership;
                };

                struct DataWriterQos {
                   DurabilityQosPolicy durability;
                   DeadlineQosPolicy deadline;
                   LatencyBudgetQosPolicy latency_budget;
                   LivelinessQosPolicy liveliness;
                   ReliabilityQosPolicy reliability;
```

```
         DestinationOrderQosPolicy destination_order;
         HistoryQosPolicy history;
         ResourceLimitsQosPolicy resource_limits;
         TransportPriorityQosPolicy transport_priority;
         LifespanQosPolicy lifespan;
         UserDataQosPolicy user_data;
         OwnershipQosPolicy ownership;
         OwnershipStrengthQosPolicy ownership_strength;
         WriterDataLifecycleQosPolicy writer_data_lifecycle;
      };

      struct PublisherQos {
         PresentationQosPolicy presentation;
         PartitionQosPolicy partition;
         GroupDataQosPolicy group_data;
         EntityFactoryQosPolicy entity_factory;
      };

      struct DataReaderQos {
         DurabilityQosPolicy durability;
         DeadlineQosPolicy deadline;
         LatencyBudgetQosPolicy latency_budget;
         LivelinessQosPolicy liveliness;
         ReliabilityQosPolicy reliability;
         DestinationOrderQosPolicy destination_order;
         HistoryQosPolicy history;
         ResourceLimitsQosPolicy resource_limits;
         UserDataQosPolicy user_data;
         OwnershipQosPolicy ownership;
         TimeBasedFilterQosPolicy time_based_filter;
         ReaderDataLifecycleQosPolicy reader_data_lifecycle;
      };

      struct SubscriberQos {
         PresentationQosPolicy presentation;
         PartitionQosPolicy partition;
         GroupDataQosPolicy group_data;
         EntityFactoryQosPolicy entity_factory;
      };

      // -------------------------------------------------------
      struct ParticipantBuiltinTopicData {
         BuiltinTopicKey_t key;
         UserDataQosPolicy user_data;
      };

      struct TopicBuiltinTopicData {
         BuiltinTopicKey_t key;
         string name;
         string type_name;
```

```
            DurabilityQosPolicy durability;
            DeadlineQosPolicy deadline;
            LatencyBudgetQosPolicy latency_budget;
            LivelinessQosPolicy liveliness;
            ReliabilityQosPolicy reliability;
            TransportPriorityQosPolicy transport_priority;
            LifespanQosPolicy lifespan;
            DestinationOrderQosPolicy destination_order;
            HistoryQosPolicy history;
            ResourceLimitsQosPolicy resource_limits;
            OwnershipQosPolicy ownership;
            TopicDataQosPolicy topic_data;
         };

         struct PublicationBuiltinTopicData {
            BuiltinTopicKey_t key;
            BuiltinTopicKey_t participant_key;
            string topic_name;
            string type_name;
            DurabilityQosPolicy durability;
            DeadlineQosPolicy deadline;
            LatencyBudgetQosPolicy latency_budget;
            LivelinessQosPolicy liveliness;
            ReliabilityQosPolicy reliability;
            LifespanQosPolicy lifespan;
            UserDataQosPolicy user_data;
            OwnershipStrengthQosPolicy ownership_strength;
            PresentationQosPolicy presentation;
            PartitionQosPolicy partition;
            TopicDataQosPolicy topic_data;
            GroupDataQosPolicy group_data;
         };

         struct SubscriptionBuiltinTopicData {
            BuiltinTopicKey_t key;
            BuiltinTopicKey_t participant_key;
            string topic_name;
            string type_name;
            DurabilityQosPolicy durability;
            DeadlineQosPolicy deadline;
            LatencyBudgetQosPolicy latency_budget;
            LivelinessQosPolicy liveliness;
            ReliabilityQosPolicy reliability;
            DestinationOrderQosPolicy destination_order;
            UserDataQosPolicy user_data;
            TimeBasedFilterQosPolicy time_based_filter;
            PresentationQosPolicy presentation;
            PartitionQosPolicy partition;
            TopicDataQosPolicy topic_data;
            GroupDataQosPolicy group_data;
```

```
  };

  // -----------------------------------------------------
  interface Entity {
  //  ReturnCode_t
  //  set_qos(
  //      in EntityQos qos);
  //
  //  ReturnCode_t
  //  get_qos(
  //      inout EntityQos qos);
  //
  //  ReturnCode_t
  //  set_listener(
  //      in Listener l,
  //      in StatusMask mask);
  //
  //  Listener
  //  get_listener();
  ReturnCode_t
  enable();
  StatusCondition
  get_statuscondition();
  StatusMask
  get_status_changes();
  };

  // -----------------------------------------------------
  interface DomainParticipant : Entity {

      // Factory interfaces
  Publisher
  create_publisher(
      in PublisherQos qos,
      in PublisherListener a_listener,
      in StatusMask mask);

  ReturnCode_t
  delete_publisher(
      in Publisher p);

  Subscriber
  create_subscriber(
      in SubscriberQos qos,
      in SubscriberListener a_listener,
      in StatusMask mask);

  ReturnCode_t
  delete_subscriber(
      in Subscriber s);
```

PRISMTECH

```
Subscriber
get_builtin_subscriber();

Topic
create_topic(
    in string topic_name,
    in string type_name,
    in TopicQos qos,
    in TopicListener a_listener,
    in StatusMask mask);

ReturnCode_t
delete_topic(
    in Topic a_topic);

Topic
find_topic(
    in string topic_name,
    in Duration_t timeout);

TopicDescription
lookup_topicdescription(
    in string name);

ContentFilteredTopic
create_contentfilteredtopic(
    in string name,
    in Topic related_topic,
    in string filter_expression,
    in StringSeq expression_parameters);

ReturnCode_t
delete_contentfilteredtopic(
    in ContentFilteredTopic a_contentfilteredtopic);

MultiTopic
create_multitopic(
    in string name,
    in string type_name,
    in string subscription_expression,
    in StringSeq expression_parameters);

ReturnCode_t
delete_multitopic(
    in MultiTopic a_multitopic);

ReturnCode_t
delete_contained_entities();
```

```
ReturnCode_t
set_qos(
    in DomainParticipantQos qos);

ReturnCode_t
get_qos(
    inout DomainParticipantQos qos);

ReturnCode_t
set_listener(
    in DomainParticipantListener a_listener,
    in StatusMask mask);
DomainParticipantListener
get_listener();

ReturnCode_t
ignore_participant(
    in InstanceHandle_t handle);

ReturnCode_t
ignore_topic(
    in InstanceHandle_t handle);

ReturnCode_t
ignore_publication(
    in InstanceHandle_t handle);

ReturnCode_t
ignore_subscription(
    in InstanceHandle_t handle);

DomainId_t
get_domain_id();

ReturnCode_t
assert_liveliness();
ReturnCode_t
set_default_publisher_qos(
    in PublisherQos qos);

ReturnCode_t
get_default_publisher_qos(
    inout PublisherQos qos);

ReturnCode_t
set_default_subscriber_qos(
    in SubscriberQos qos);

ReturnCode_t
get_default_subscriber_qos(
```

```
                inout SubscriberQos qos);

        ReturnCode_t
        set_default_topic_qos(
            in TopicQos qos);

        ReturnCode_t
        get_default_topic_qos(
            inout TopicQos qos);

         boolean
        contains_entity(
            in InstanceHandle_t a_handle);

        ReturnCode_t
        get_current_time(
            inout Time_t current_time);
    };

         interface DomainParticipantFactory {
         //
         //   DomainParticipantFactory
         //   get_instance();
         //

        DomainParticipant
        create_participant(
            in DomainId_t domainId,
            in DomainParticipantQos qos,
            in DomainParticipantListener a_listener,
            in StatusMask mask);

        ReturnCode_t
        delete_participant(
            in DomainParticipant a_participant);

        DomainParticipant
        lookup_participant(
            in DomainId_t domainId);

        ReturnCode_t
        set_default_participant_qos(
            in DomainParticipantQos qos);

        ReturnCode_t
        get_default_participant_qos(
            inout DomainParticipantQos qos);

        ReturnCode_t
        set_qos(
```

```
              in DomainParticipantFactoryQos qos);

    ReturnCode_t
    get_qos(
        inout DomainParticipantFactoryQos qos);

    ReturnCode_t
       delete_domain
          (in Domain a_domain);

    Domain
       lookup_domain
          (in DomainId_t domainId);

    ReturnCode_t
       create_persistent_snapshot(
          in string partition_expression,
          in string topic_expression,
          in string URI);

    ReturnCode_t
       delete_contained_entities();
    };

    interface TypeSupport {
    //  ReturnCode_t
    //  register_type(
    //      in DomainParticipant domain,
    //      in string type_name);
    //
    //  string
    //  get_type_name();
    };

    // --------------------------------------------------------
    interface TopicDescription {
    string
    get_type_name();
    string
    get_name();
    DomainParticipant

    get_participant();
    };

    interface Topic : Entity, TopicDescription {
    ReturnCode_t
    set_qos(
        in TopicQos qos);
```

```
ReturnCode_t
get_qos(
    inout TopicQos qos);

ReturnCode_t
set_listener(
    in TopicListener a_listener,
    in StatusMask mask);

TopicListener
get_listener();

// Access the status
ReturnCode_t
get_inconsistent_topic_status(
    inout InconsistentTopicStatus a_status);

ReturnCode_t
get_all_data_disposed_topic_status(
    inout AllDataDisposedTopicStatus a_status);
 };

interface ContentFilteredTopic : TopicDescription {
string
get_filter_expression();

ReturnCode_t
get_expression_parameters(
    inout StringSeq expression_parameters);

ReturnCode_t
set_expression_parameters(
    in StringSeq expression_parameters);

Topic
get_related_topic();
 };

 interface MultiTopic : TopicDescription {

stringget_subscription_expression();

ReturnCode_t
get_expression_parameters(
    inout StringSeq expression_parameters);

ReturnCode_t
set_expression_parameters(
    in StringSeq expression_parameters);
 };
```

```
 // ----------------------------------------------------
 interface Publisher : Entity {

DataWriter
create_datawriter(
    in Topic a_topic,
    in DataWriterQos qos,
    in DataWriterListener a_listener,
    in StatusMask mask);

ReturnCode_t
delete_datawriter(
    in DataWriter a_datawriter);

DataWriter
lookup_datawriter(
    in string topic_name);

ReturnCode_t
delete_contained_entities();

ReturnCode_t
set_qos(
    in PublisherQos qos);

ReturnCode_t
get_qos(
    inout PublisherQos qos);

ReturnCode_t
set_listener(
    in PublisherListener a_listener,
    in StatusKindMask mask);

PublisherListener
get_listener();

ReturnCode_t
suspend_publications();

ReturnCode_t
resume_publications();

ReturnCode_t
begin_coherent_changes();

ReturnCode_t
end_coherent_changes();
```

```
ReturnCode_t
wait_for_acknowledgments(
    in Duration_t max_wait);

DomainParticipant
get_participant();

ReturnCode_t
set_default_datawriter_qos(
    in DataWriterQos qos);

ReturnCode_t
get_default_datawriter_qos(
    inout DataWriterQos qos);

ReturnCode_t
copy_from_topic_qos(
    inout DataWriterQos a_datawriter_qos,
    in TopicQos a_topic_qos);
};

interface DataWriter : Entity {
//  InstanceHandle_t
//  register_instance(
//      in Data instance_data);
//
//  InstanceHandle_t
//  register_instance_w_timestamp(
//      in Data instance_data,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  unregister_instance(
//      in Data instance_data,
//      in InstanceHandle_t handle);
//
//  ReturnCode_t
//  unregister_instance_w_timestamp(
//      in Data instance_data,
//      in InstanceHandle_t handle,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  write(
//      in Data instance_data,
//      in InstanceHandle_t handle);
//
//  ReturnCode_t
//  write_w_timestamp(
//      in Data instance_data,
```

```
//      in InstanceHandle_t handle,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  dispose(
//      in Data instance_data,
//      in InstanceHandle_t instance_handle);
//
//  ReturnCode_t
//  dispose_w_timestamp(
//      in Data instance_data,
//      in InstanceHandle_t instance_handle,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  get_key_value(
//      inout Data key_holder,
//      in InstanceHandle_t handle);
//
//  InstanceHandle_t
//  lookup_instance(
//      in Data instance);

ReturnCode_t
set_qos(
    in DataWriterQos qos);

ReturnCode_t
get_qos(
    inout DataWriterQos qos);

ReturnCode_t
set_listener(
    in DataWriterListener a_listener,
    in StatusMask mask);

DataWriterListener
get_listener();

Topic
get_topic();

Publisher
get_publisher();

ReturnCode_t
wait_for_acknowledgments(
    in Duration_t max_wait);

// Access the status
```

PRISMTECH

```
ReturnCode_t
get_liveliness_lost_status(
    inout LivelinessLostStatus status);

ReturnCode_t
get_offered_deadline_missed_status(
    inout OfferedDeadlineMissedStatus status);

ReturnCode_t
get_offered_incompatible_qos_status(
    inout OfferedIncompatibleQosStatus status);

ReturnCode_t
get_publication_matched_status(
    inout PublicationMatchedStatus status);

ReturnCode_t
assert_liveliness();

ReturnCode_t
get_matched_subscriptions(
    inout InstanceHandleSeq subscription_handles);

ReturnCode_t
get_matched_subscription_data(
    inout SubscriptionBuiltinTopicData subscription_data,
    in InstanceHandle_t subscription_handle);
};

// ---------------------------------------------------

interface Subscriber : Entity {

DataReader
create_datareader(
    in TopicDescription a_topic,
    in DataReaderQos qos,
    in DataReaderListener a_listener,
    in StatusMask mask);

ReturnCode_t
delete_datareader(
    in DataReader a_datareader);

ReturnCode_t
delete_contained_entities();

DataReader
lookup_datareader(
```

```
                        in string topic_name);

            ReturnCode_t
            get_datareaders(
                inout DataReaderSeq readers,
                in SampleStateMask sample_states,
                in ViewStateMask view_states,
                in InstanceStateMask instance_states);

            ReturnCode_t
            notify_datareaders();

            ReturnCode_t
            set_qos(
                in SubscriberQos qos);

            ReturnCode_t
            get_qos(
                inout SubscriberQos qos);

            ReturnCode_t
            set_listener(
                in SubscriberListener a_listener,
                in StatusMask mask);

            SubscriberListener
            get_listener();

            ReturnCode_t
            begin_access();

            ReturnCode_t
            end_access();

            DomainParticipant
            get_participant();

            ReturnCode_t
            set_default_datareader_qos(
                in DataReaderQos qos);

            ReturnCode_t
            get_default_datareader_qos(
                inout DataReaderQos qos);

            ReturnCode_t
            copy_from_topic_qos(
                inout DataReaderQos a_datareader_qos,
                in TopicQos a_topic_qos);
        };
```

PRISMTECH

```
interface DataReader : Entity {
//  ReturnCode_t
//  read(
//      inout DataSeq data_values,
//      inout SampleInfoSeq info_seq,
//      in long max_samples,
//      in SampleStateMask sample_states,
//      in ViewStateMask view_states,
//      in InstanceStateMask instance_states);
//
//  ReturnCode_t
//  take(
//      inout DataSeq data_values,
//      inout SampleInfoSeq info_seq,
//      in long max_samples,
//      in SampleStateMask sample_states,
//      in ViewStateMask view_states,
//      in InstanceStateMask instance_states);
//
//  ReturnCode_t
//  read_w_condition(
//      inout DataSeq data_values,
//      inout SampleInfoSeq info_seq,
//      in long max_samples,
//      in ReadCondition a_condition);
//
//  ReturnCode_t
//  take_w_condition(
//      inout DataSeq data_values,
//      inout SampleInfoSeq info_seq,
//      in long max_samples,
//      in ReadCondition a_condition);
//
//  ReturnCode_t
//  read_next_sample(
//      inout Data data_value,
//      inout SampleInfo sample_info);
//
//  ReturnCode_t
//  take_next_sample(
//      inout Data data_value,
//      inout SampleInfo sample_info);
//
//  ReturnCode_t
//  read_instance(
//      inout DataSeq data_values,
//      inout SampleInfoSeq info_seq,
//      in long max_samples,
//      in InstanceHandle_t a_handle,
```

```
//       in SampleStateMask sample_states,
//       in ViewStateMask view_states,
//       in InstanceStateMask instance_states);
//
// ReturnCode_t
// take_instance(
//       inout DataSeq data_values,
//       inout SampleInfoSeq info_seq,
//       in long max_samples,
//       in InstanceHandle_t a_handle,
//       in SampleStateMask sample_states,
//       in ViewStateMask view_states,
//       in InstanceStateMask instance_states);
//
// ReturnCode_t
// read_next_instance(
//       inout DataSeq data_values,
//       inout SampleInfoSeq info_seq,
//       in long max_samples,
//       in InstanceHandle_t a_handle,
//       in SampleStateMask sample_states,
//       in ViewStateMask view_states,
//       in InstanceStateMask instance_states);
//
// ReturnCode_t
// take_next_instance(
//       inout DataSeq data_values,
//       inout SampleInfoSeq info_seq,
//       in long max_samples,
//       in InstanceHandle_t a_handle,
//       in SampleStateMask sample_states,
//       in ViewStateMask view_states,
//       in InstanceStateMask instance_states);
//
// ReturnCode_t
// read_next_instance_w_condition(
//       inout DataSeq data_values,
//       inout SampleInfoSeq info_seq,
//       in long max_samples,
//       in InstanceHandle_t a_handle,
//       in ReadCondition a_condition);
//
// ReturnCode_t
// take_next_instance_w_condition(
//       inout DataSeq data_values,
//       inout SampleInfoSeq info_seq,
//       in long max_samples,
//       in InstanceHandle_t a_handle,
//       in ReadCondition a_condition);
//
```

```
//  ReturnCode_t
//  return_loan(
//      inout DataSeq data_values,
//      inout SampleInfoSeq info_seq);
//
//  ReturnCode_t
//  get_key_value(
//      inout Data key_holder,
//      in InstanceHandle_t handle);

ReadCondition
create_readcondition(
    in SampleStateMask sample_states,
    in ViewStateMask view_states,
    in InstanceStateMask instance_states);

QueryCondition
create_querycondition(
    in SampleStateMask sample_states,
    in ViewStateMask view_states,
    in InstanceStateMask instance_states,
    in string query_expression,
    in StringSeq query_parameters);

ReturnCode_t
delete_readcondition(
    in ReadCondition a_condition);

ReturnCode_t
delete_contained_entities();

ReturnCode_t
set_qos(
    in DataReaderQos qos);

ReturnCode_t
get_qos(
    inout DataReaderQos qos);

ReturnCode_t
set_listener(
    in DataReaderListener a_listener,
    in StatusMask mask);

DataReaderListener
get_listener();

TopicDescription
get_topicdescription();
```

```
Subscriber
get_subscriber();

ReturnCode_t
get_sample_rejected_status(
    inout SampleRejectedStatus status);

ReturnCode_t
get_liveliness_changed_status(
    inout LivelinessChangedStatus status);

ReturnCode_t
get_requested_deadline_missed_status(
    inout RequestedDeadlineMissedStatus status);

ReturnCode_t
get_requested_incompatible_qos_status(
    inout RequestedIncompatibleQosStatus status);

ReturnCode_t
get_subscription_matched_status(
    inout SubscriptionMatchedStatus status);

ReturnCode_t
get_sample_lost_status(
    inout SampleLostStatus status);

ReturnCode_t
wait_for_historical_data(
    in Duration_t max_wait);

ReturnCode_t
get_matched_publications(
    inout InstanceHandleSeq publication_handles);

ReturnCode_t
get_matched_publication_data(
    inout PublicationBuiltinTopicData publication_data,
    in InstanceHandle_t publication_handle);
};

struct SampleInfo {
    SampleStateKind sample_state;
    ViewStateKind view_state;
    InstanceStateKind instance_state;
    Time_t source_timestamp;
    InstanceHandle_t instance_handle;
    BuiltinTopicKey_t publication_handle;
    long disposed_generation_count;
    long no_writers_generation_count;
```

PRISMTECH

```
            long sample_rank;
            long generation_rank;
            long absolute_generation_rank;
            boolean valid_data;
         };
        typedef sequence<SampleInfo> SampleInfoSeq;
};
```

Foo.idl

```
        // Implied IDL for type "Foo"
        // Example user defined structure

        struct Foo {
           long dummy;
        };

        typedef sequence<Foo> FooSeq;

        #include "dds_dcps.idl"

      interface FooTypeSupport : DDS::TypeSupport {
      DDS::ReturnCode_t
      register_type(
          in DDS::DomainParticipant participant,
          in string type_name);

      string
      get_type_name();
      };

       interface FooDataWriter : DDS::DataWriter {
      DDS::InstanceHandle_t
      register_instance(
          in Foo instance_data);

      DDS::InstanceHandle_t
      register_instance_w_timestamp(
          in Foo instance_data,
          in DDS::InstanceHandle_t handle,
          in DDS::Time_t source_timestamp);

      DDS::ReturnCode_t
      unregister_instance(
          in Foo instance_data,
          in DDS::InstanceHandle_t handle);

      DDS::ReturnCode_t
      unregister_instance_w_timestamp(
          in Foo instance_data,
```

```
    in DDS::InstanceHandle_t handle,
    in DDS::Time_t source_timestamp);

DDS::ReturnCode_t
write(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle);

DDS::ReturnCode_t
write_w_timestamp(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle,
    in DDS::Time_t source_timestamp);

DDS::ReturnCode_t
dispose(
    in Foo instance_data,
    in DDS::InstanceHandle_t instance_handle);

DDS::ReturnCode_t
dispose_w_timestamp(
    in Foo instance_data,
    in DDS::InstanceHandle_t instance_handle,
    in DDS::Time_t source_timestamp);

DDS::ReturnCode_t
get_key_value(
    inout Foo key_holder,
    in DDS::InstanceHandle_t handle);

DDS::InstanceHandle_t
lookup_instance(
    in Foo instance_data);
};
interface FooDataReader : DDS::DataReader {

DDS::ReturnCode_t
read(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in long max_samples,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);

DDS::ReturnCode_t
take(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in long max_samples,
```

```
        in DDS::SampleStateMask sample_states,
        in DDS::ViewStateMask view_states,
        in DDS::InstanceStateMask instance_states);

    DDS::ReturnCode_t
    read_w_condition(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::ReadCondition a_condition);

    DDS::ReturnCode_t
    take_w_condition(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::ReadCondition a_condition);

    DDS::ReturnCode_t
    read_next_sample(
        inout Foo data_value,
        inout DDS::SampleInfo sample_info);

    DDS::ReturnCode_t
    take_next_sample(
        inout Foo data_value,
        inout DDS::SampleInfo sample_info);

    DDS::ReturnCode_t
    read_instance(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::InstanceHandle_t a_handle,
        in DDS::SampleStateMask sample_states,
        in DDS::ViewStateMask view_states,
        in DDS::InstanceStateMask instance_states);

    DDS::ReturnCode_t
    take_instance(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::InstanceHandle_t a_handle,
        in DDS::SampleStateMask sample_states,
        in DDS::ViewStateMask view_states,
        in DDS::InstanceStateMask instance_states);

    DDS::ReturnCode_t
    read_next_instance(
```

```
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::InstanceHandle_t a_handle,
        in DDS::SampleStateMask sample_states,
        in DDS::ViewStateMask view_states,
        in DDS::InstanceStateMask instance_states);

DDS::ReturnCode_t
take_next_instance(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::InstanceHandle_t a_handle,
        in DDS::SampleStateMask sample_states,
        in DDS::ViewStateMask view_states,
        in DDS::InstanceStateMask instance_states);

DDS::ReturnCode_t
read_next_instance_w_condition(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::InstanceHandle_t a_handle,
        in DDS::ReadCondition a_condition);

DDS::ReturnCode_t
take_next_instance_w_condition(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq,
        in long max_samples,
        in DDS::InstanceHandle_t a_handle,
        in DDS::ReadCondition a_condition);

DDS::ReturnCode_t
return_loan(
        inout FooSeq data_values,
        inout DDS::SampleInfoSeq info_seq);

DDS::ReturnCode_t
get_key_value(
        inout Foo key_holder,
        in DDS::InstanceHandle_t handle);

DDS::InstanceHandle_t
lookup_instance(
        in Foo instance);
    };
```

D

SampleStates, ViewStates and InstanceStates

Data is made available to the application by the following operations on `DataReader` objects: `read` and `take` operations. The general semantics of the `read` operations is that the application only gets access to the matching data; the data remain available in the Data Distribution Services and can be read again. The semantics of the `take` operations is that the data is not available in the Data Distribution Service; that data will no longer be accessible to the `DataReader`. Consequently, it is possible for a `DataReader` to access the same sample multiple times but only if all previous accesses were `read` operations.

Each of these operations returns an ordered collection of `Data` values and associated `SampleInfo` 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 `HistoryQosPolicy` allow for it.

SampleInfo Class

`SampleInfo` is the information that accompanies each sample that is 'read' or 'taken'. It contains, among others, the following information:

- The `sample_state` (`READ_SAMPLE_STATE` or `NOT_READ_SAMPLE_STATE`)

- The `view_state`, (`NEW_VIEW_STATE` or `NOT_NEW_VIEW_STATE`)

- The `instance_state` (`ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE` or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`)

sample_state

For each sample, the Data Distribution Service internally maintains a `sample_state` specific to each `DataReader`. The `sample_state` can either be `READ_SAMPLE_STATE` or `NOT_READ_SAMPLE_STATE`.

`READ_SAMPLE_STATE` indicates that the `DataReader` has already accessed that sample by means of `read`. Had the sample been accessed by `take` it would no longer be available to the `DataReader`;

- NOT_READ_SAMPLE_STATE indicates that the DataReader has not accessed that sample before.
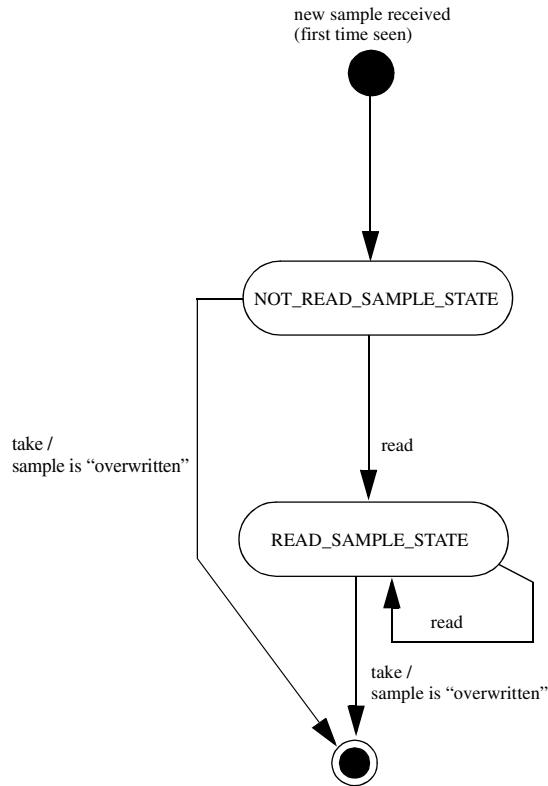
Figure 20 sample_state for a Single Sample State Chart

State per Sample

The sample_state available in the SampleInfo reflect the sample_state of each sample. The sample_state can be: different for all samples in the returned collection that refer to the same instance.

instance_state

For each instance the Data Distribution Service internally maintains an instance_state. The instance_state can be:

- ALIVE_INSTANCE_STATE indicates that

  - samples have been received for the instance

  - and there are live DataWriter objects writing the instance

- and the instance has not been explicitly disposed of (or else samples have been received after it was disposed of).

- `NOT_ALIVE_DISPOSED_INSTANCE_STATE` indicates the instance was disposed of by a `DataWriter`, either explicitly by means of the `dispose` operation or implicitly in case the `autodispose_unregistered_instances` field of the `WriterDataLyfecycleQosPolicy` equals `TRUE` when the instance gets unregistered (see Section 3.1.3.23, *WriterDataLifecycleQosPolicy*), and no new samples for that instance have been written afterwards

- `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` indicates the instance has been declared as not-alive by the `DataReader` because it detected that there are no live `DataWriter` objects writing that instance.

OwnershipQosPolicy

The precise events that cause the `instance_state` to change depends on the setting of the `OwnershipQosPolicy`:

- If `OwnershipQosPolicy` is set to `EXCLUSIVE_OWNERSHIP_QOS`, then the `instance_state` becomes `NOT_ALIVE_DISPOSED_INSTANCE_STATE` only if the `DataWriter` that "owns" the instance explicitly disposes of it. The `instance_state` becomes `ALIVE_INSTANCE_STATE` again only if the `DataWriter` that owns the instance writes it;

- If `OwnershipQosPolicy` is set to `SHARED_OWNERSHIP_QOS`, then the `instance_state` becomes `NOT_ALIVE_DISPOSED_INSTANCE_STATE` if any `DataWriter` explicitly disposes of the instance. The `instance_state` becomes `ALIVE_INSTANCE_STATE` as soon as any `DataWriter` writes the instance again.
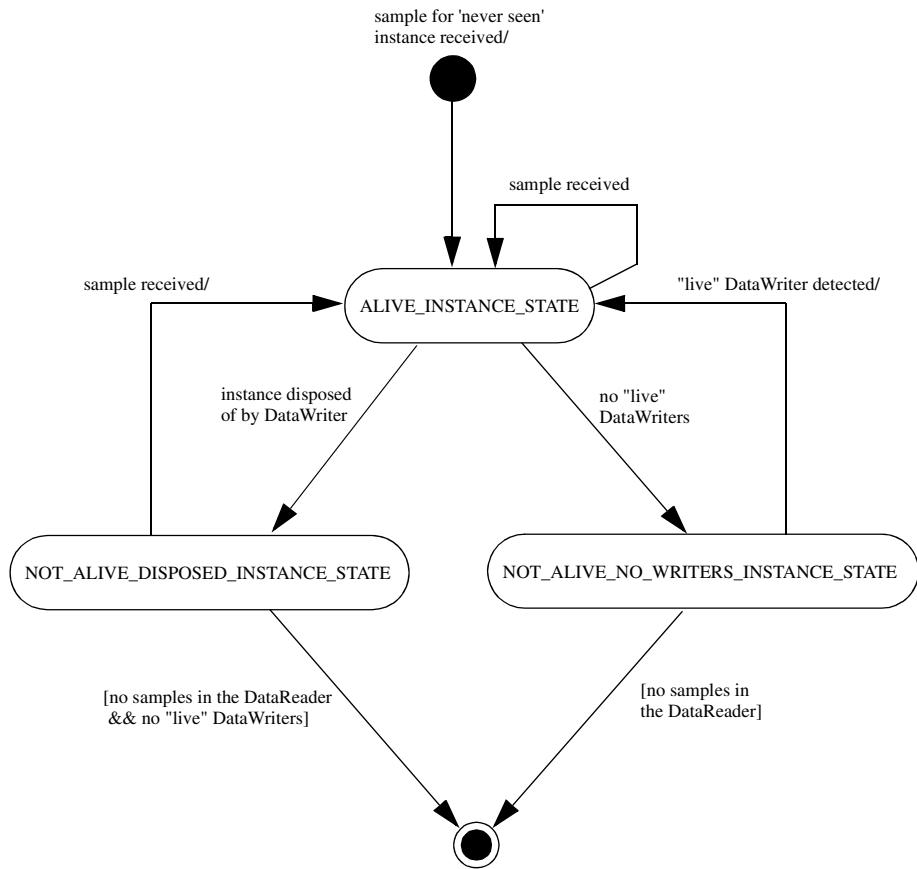
Figure 21 State Chart of the instance_state for a Single Instance

Snapshot

The `instance_state` available in the `SampleInfo` 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.

view_state

For each instance (identified by the `key`), the Data Distribution Service internally maintains a `view_state` relative to each `DataReader`. The `view_state` can either be `NEW_VIEW_STATE` or `NOT_NEW_VIEW_STATE`.

- NEW_VIEW_STATE indicates that either this is the first time that the DataReader has ever accessed samples of that instance, or else that the DataReader has accessed previous samples of the instance, but the instance has since been reborn (i.e. becomes not-alive and then alive again)

- NOT_NEW_VIEW_STATE indicates that the DataReader has already accessed samples of the same instance and that the instance has not been reborn since
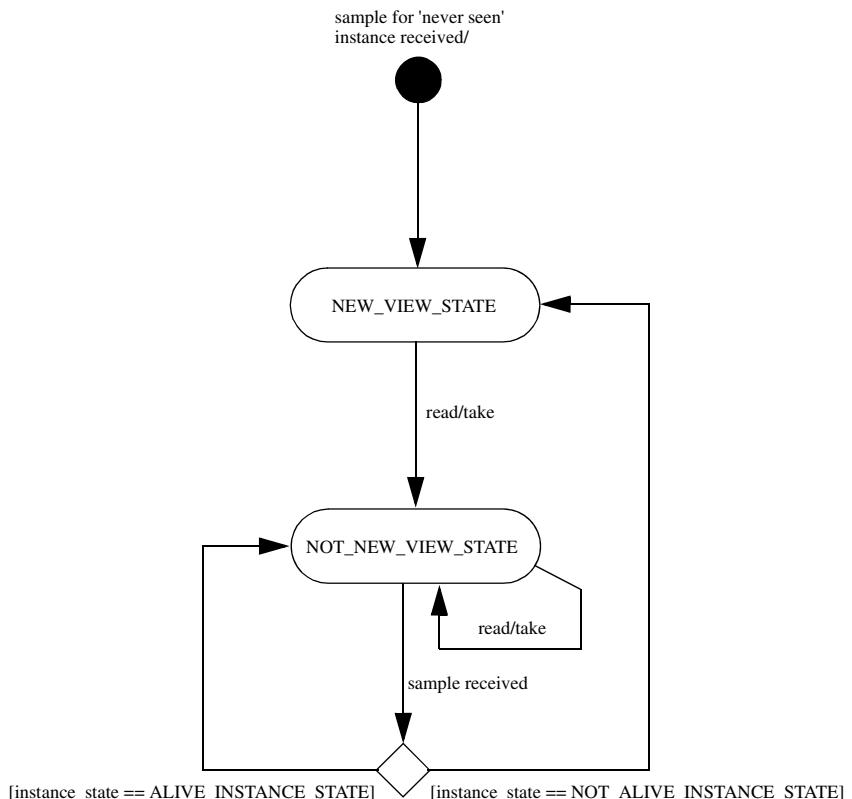
Figure 22 view_state for a Single Instance State Chart

Snapshot

The view_state available in the SampleInfo is a snapshot of view_state of the instance relative to the DataReader 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.

State Masks

State Definitions

All states are available as a constant. These convenience constants can be used to create a bit mask (e.g. to be used as operation parameters) by performing an AND or OR operation. They can also be used for testing whether a state is set.

The sample state definitions indicates whether or not the matching data sample has already been read:

- `READ_SAMPLE_STATE`: sample has already been read
- `NOT_READ_SAMPLE_STATE`: sample has not been read

The view state definitions indicates whether the `DataReader` has already seen samples for the most-current generation of the related instance

- `NEW_VIEW_STATE`: all samples of this instance are new
- `NOT_NEW_VIEW_STATE`: some or all samples of this instance are not new

The instance state definitions indicates whether the instance is currently in existence or, if it has been disposed of, the reason why it was disposed of:

- `ALIVE_INSTANCE_STATE`: this instance is currently in existence
- `NOT_ALIVE_DISPOSED_INSTANCE_STATE`: this instance was disposed of by a `DataWriter`
- `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`: the instance has been disposed of by the `DataReader` because none of the `DataWriter` objects currently "alive" (according to the `LivelinessQosPolicy`) are writing the instance.

Pre-defined Bit Mask Definitions

For convenience, some pre-defined bit masks are available as a constant definition. These bit mask constants can be used where a state bit mask is required. They can also be used for testing whether certain bits are set.

The sample state bit mask definition selects both sample states

- `ANY_SAMPLE_STATE`: either the sample has already been read or not read

The view state bit mask definition selects both view states

- `ANY_VIEW_STATE`: either the sample has already been seen or not seen

The instance state bit mask definitions selects a combination of instance states

- `NOT_ALIVE_INSTANCE_STATE`: this instance was disposed of by a `DataWriter` or the `DataReader`
- `ANY_INSTANCE_STATE`: this instance is either in existence or not in existence

Operations Concerning States

The application accesses data by means of the operations `read` or `take` on the `DataReader`. These operations return an ordered collection of `DataSamples` consisting of a `SampleInfo` part and a `Data` part. The way the Data Distribution Service builds this collection (i.e., the data-samples that are parts of the list as well as their order) depends on `QosPolicy` settings set on the `DataReader` and the `Subscriber`, as well as the source timestamp of the samples and the parameters passed to the `read`/`take` operations, namely:

- the desired sample states (i.e., `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE`, or `ANY_SAMPLE_STATE`)

- the desired view states (i.e., `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE`, or `ANY_VIEW_STATE`)

- the desired instance states (`ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`, `NOT_ALIVE_INSTANCE_STATE`, or `ANY_INSTANCE_STATE`)

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

On output, the collection of `Data` values and the collection of `SampleInfo` objects are of the same length and are in a one-to-one correspondence. Each `SampleInfo` provides information, such as the `source_timestamp`, the `sample_state`, `view_state`, and `instance_state`, etc., about the matching sample.

Some elements in the returned collection may not have valid data. If the `instance_state` in the `SampleInfo` is `NOT_ALIVE_DISPOSED_INSTANCE_STATE` or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`, then the last sample for that instance in the collection, that is, the one whose `SampleInfo` has `sample_rank==0` does not contain valid data. Samples that contain no data do not count towards the limits imposed by the `ResourceLimitsQosPolicy`.

read

The act of reading a sample sets its `sample_state` to `READ_SAMPLE_STATE`. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

take

The act of taking a sample removes it from the `DataReader` 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 instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

read_w_condition

In case the `ReadCondition` is a 'plain' `ReadCondition` and not the specialized `QueryCondition`, the operation is equivalent to calling `read` and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in the `ReadCondition`. Using this operation the application can avoid repeating the same parameters specified when creating the `ReadCondition`.

take_w_condition

The act of taking a sample removes it from the `DataReader` 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 instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

In case the `ReadCondition` is a 'plain' `ReadCondition` and not the specialized `QueryCondition`, the operation is equivalent to calling `take` and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in the `ReadCondition`. Using this operation the application can avoid repeating the same parameters specified when creating the `ReadCondition`.

read_next_sample

The `read_next_sample` operation is semantically equivalent to the `read` operation where the input `Data` sequence has `max_len=1`, the `sample_states=NOT_READ_SAMPLE_STATE`, the `view_states=ANY_VIEW_STATE`, and the `instance_states=ANY_INSTANCE_STATE`.

take_next_sample

The `take_next_sample` operation is semantically equivalent to the `take` operation where the input sequence has `max_len=1`, the `sample_states=NOT_READ_SAMPLE_STATE`, the `view_states=ANY_VIEW_STATE`, and the `instance_states=ANY_INSTANCE_STATE`.

read_instance

The act of reading a sample sets its `sample_state` to `READ_SAMPLE_STATE`. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

take_instance

The act of taking a sample removes it from the `DataReader` 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 instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

Appendices

E

Interface Inheritance

This appendix gives an overview of the inheritance relations of the DCPS interfaces.
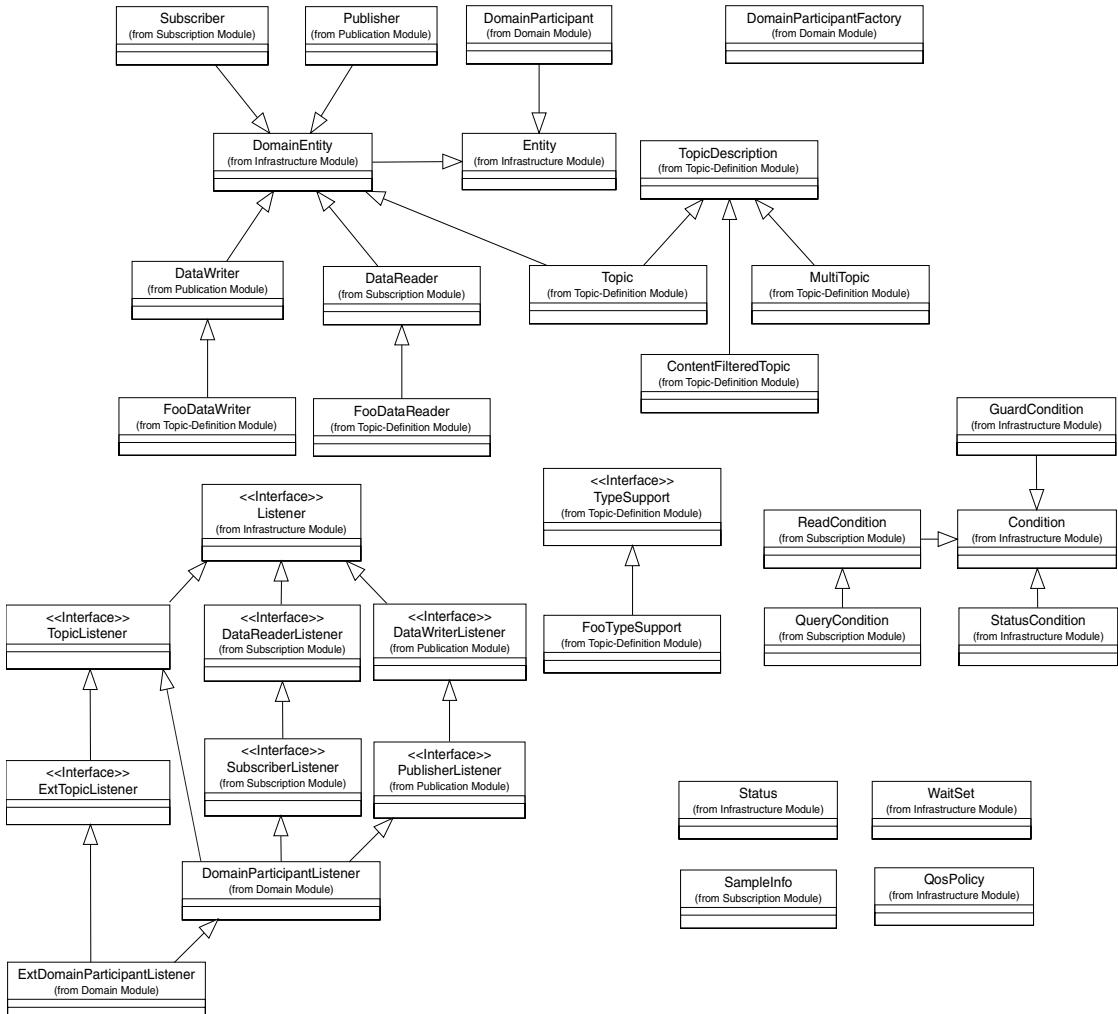
Figure 23 DCPS Inheritance

PRISMTECH

F *Listeners, Conditions and Waitsets*

`Listeners` and `Conditions` (`Conditions` in conjunction with `WaitSets`) are two mechanisms that allow the application to be made aware of changes in the communication status. `Listeners` provide an event-based mechanism for the Data Distribution Service to asynchronously alert the application of the occurrence of relevant status changes. `Conditions` in conjunction with `WaitSets` provide a state-based mechanism for the Data Distribution Service to synchronously communicate the relevant status changes to the application.

Both mechanisms are based on the communication statuses associated with an `Entity` object. Not all statuses are applicable to all `Entity` objects. Which status is applicable to which `Entity` object is listed in the next table:

Table 24 Communication Statuses

Entity	Status Name	Description
Topic	INCONSISTENT_TOPIC_STATUS	Another `Topic` exists with the same name but with different characteristics.
	ALL_DATA_DISPOSED_TOPIC_STATUS	All instances of the `Topic` have been disposed by the `dispose_all_data` operation on that topic.
Subscriber	DATA_ON_READERS_STATUS	New information is available.
DataReader	SAMPLE_REJECTED_STATUS	A (received) sample has been rejected.
	LIVELINESS_CHANGED_STATUS	The liveliness of one or more `DataWriter` objects, that were writing instances read through the `DataReader` objects has changed. Some `DataWriter` object have become "active" or "inactive".
	REQUESTED_DEADLINE_MISSED_STATUS	The deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS_STATUS	A QosPolicy setting was incompatible with what is offered.
	DATA_AVAILABLE_STATUS	New information is available.
	SAMPLE_LOST_STATUS	A sample has been lost (never received).
	SUBSCRIPTION_MATCH_STATUS	The `DataReader` has found a `DataWriter` that matches the `Topic` and has compatible QoS.
DataWriter	LIVELINESS_LOST_STATUS	The liveliness that the `DataWriter` has committed through its `LivelinessQosPolicy` was not respected; thus `DataReader` objects will consider the `DataWriter` as no longer "active".
	OFFERED_DEADLINE_MISSED_STATUS	The deadline that the `DataWriter` has committed through its `DeadlineQosPolicy` was not respected for a specific instance.
	OFFERED_INCOMPATIBLE_QOS_STATUS	A QosPolicy setting was incompatible with what was requested.
	PUBLICATION_MATCH_STATUS	The `DataWriter` has found `DataReader` that matches the `Topic` and has compatible QoS.

PRISMTECH

The statuses may be classified in:

- *read communication statuses*: i.e., those that are related to arrival of data, namely DATA_ON_READERS and DATA_AVAILABLE;
- *plain communication statuses*: i.e., all the others.

For each plain communication status, there is a corresponding status class. The information from in instance of this class can be retrieved with the operations get_<status_name>_status. For example, to get the INCONSISTENT_TOPIC status (which information is stored in the InconsistentTopicStatus object), the application must call the operation get_inconsistent_topic_status. A plain communication status can only be read from the Entity on which it is applicable. For the read communication statuses there is no object available to the application.

Communication Status Event

Conceptually associated with each Entity 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 (there is no actual read-operation to read the StatusChangedFlag). The StatusChangedFlag is only conceptually needed to explain the behaviour of a Listener, therefore, it is not important whether this flag actually exists. A Listener will only be activated when the StatusChangedFlag changes from false to true (provided the Listener is attached and enabled for this particular status). The conditions which cause the StatusChangedFlag to change is slightly different for the plain communication status and the read 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 false each time the application accesses the plain communication status via the proper get_<status_name>_status operation on the Entity.

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_<status_name>_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, i.e. a listener with value null. Such a listener is treated as a NOOP[1] for all statuses activated in its bitmask and the act of calling this 'nil' listener does not reset the corresponding communication statuses.

---

1. Short for **No-Op**eration, an instruction that peforms nothing at all.
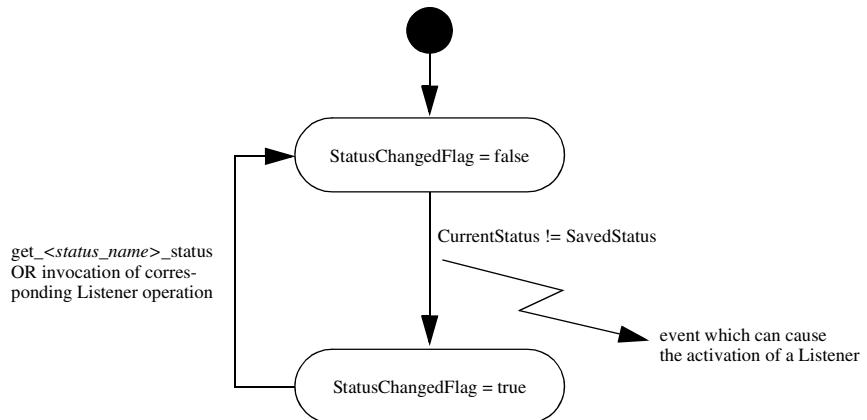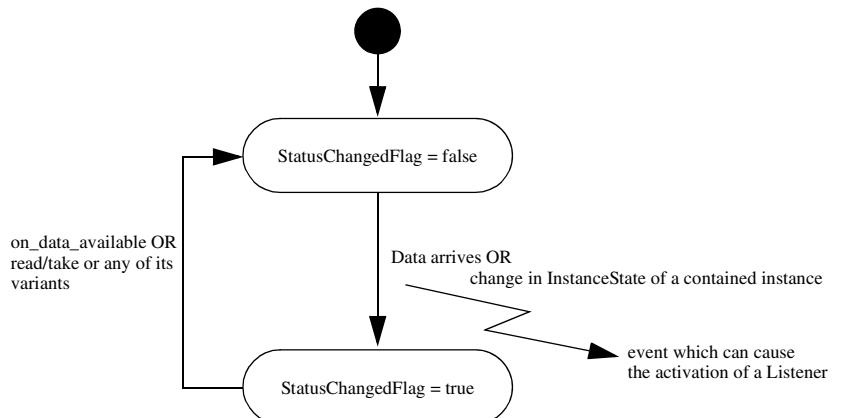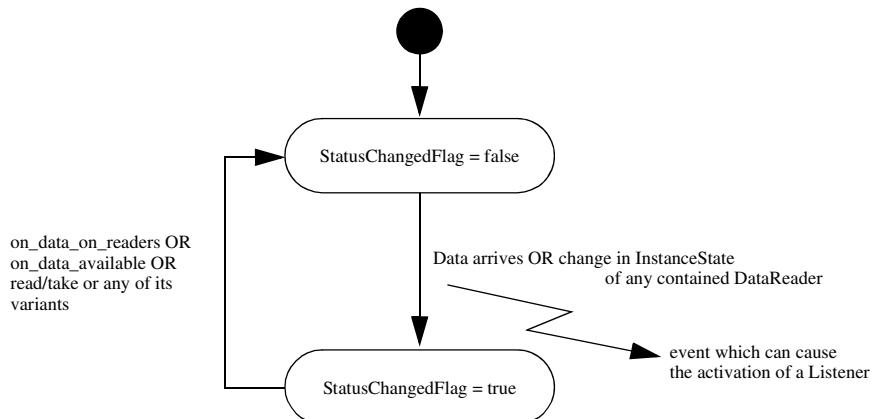
Figure 24 Plain Communication Status State Chart

For example, the value of the `StatusChangedFlag` associated with the `RequestedDeadlineMissedStatus` will become `true` each time a new deadline passes (which increases the `total_count` field within `RequestedDeadlineMissedStatus`). The value changes to `false` when the application accesses the status via the corresponding `get_requested_deadline_missed_status` operation on the proper `Entity`, or when the the `on_requested_deadline_missed` operation on the `Listener` attached to this `Entity` or one its containing entities is invoked.

For the read communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` when data arrives, or when the `InstanceState` of a contained instance changes. This can be caused by either:

- The arrival of the notification that an instance has been disposed by:

  - the `DataWriter` that owns it if its `OwnershipQosPolicyKind = EXCLUSIVE_OWNERSHIP_QOS`

  - or by any `DataWriter` if its `OwnershipQosPolicyKind = SHARED_OWNERSHIP_QOS`.

- The loss of liveliness of the `DataWriter` of an instance for which there is no other `DataWriter`.

- The arrival of the notification that an instance has been unregistered by the only `DataWriter` that is known to be writing the instance.

Figure 25 Read Communication Status DataReader Statecraft

Figure 26 Subscriber Statecraft for a Read Communication Status

- The status flag of the DATA_ON_READERS_STATUS becomes FALSE when any of the following events occurs:

  - The corresponding listener operation (on_data_on_readers) is called on the corresponding Subscriber.

  - The on_data_available listener operation is called on any DataReader belonging to the Subscriber.

  - The read or take operation (or any of its variants) is called on any DataReader belonging to the Subscriber.

Listeners

The `Listeners` provide for an event-based mechanism to asynchronous inform the application of a status change event. `Listeners` are applicable for both the read communication statuses and the plain communication statuses. When one of these status change events occur, the associated `Listener` is activated, provided some pre-conditions are satisfied. When the `Listener` is activated, it will call the corresponding `on_<status_name>` operation of that `Listener`. Each `on_<status_name>` operation available in the `Listener` of an `Entity` is also available in the `Listener` of the factory of the `Entity`.

For both the read communication statuses and the plain communication statuses a `Listener` is only activated when a `Listener` is attached to this particular `Entity` and enabled for this particular status. Statuses are enabled according the to the `StatusMask` parameter that was passed at creation time of the `Entity`, or that was passed to the `set_listener` operation.

When an event occurs for a particular `Entity` and for a particular status, but the applicable `Listener` is not activated for this status, the status is propagated up to the factory of this `Entity`. For this factory, the same propagation rules apply. When even the `DomainParticipantListener` is not attached or enabled for this status, the application will not be notified about this event. This means that a status change on a contained `Entity` only invokes the `Listener` of its factory if the `Listener` of the contained `Entity` itself does not handle the trigger event generated by the status change.

PRISMTECH
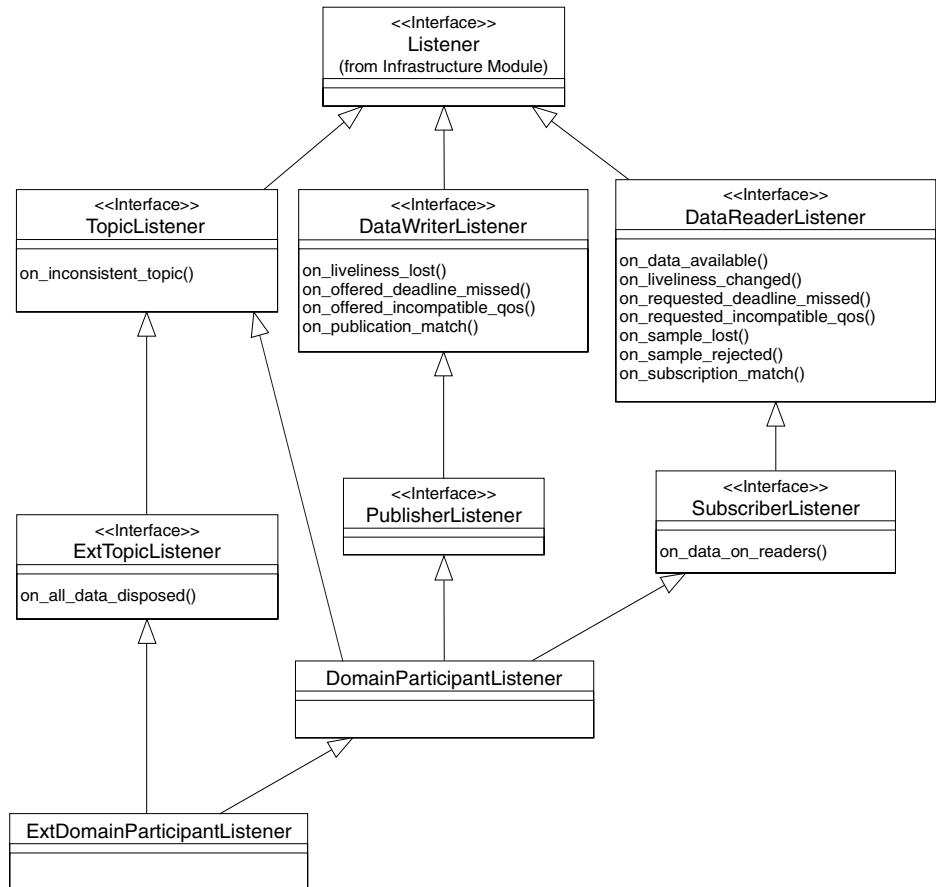
Figure 27 DCPS Listeners

The event propagation is also applicable to the read communication statuses. However, since the event here is the arrival of data, both the DATA_ON_READERS and DATA_AVAILABLE status are true. The Data Distribution Service will first attempt to handle the DATA_ON_READERS status and try to activate the SubscriberListener. When this Listener is not activated for this status the event will propagate to the DomainParticipantListener. Only when the DATA_ON_READERS status can not be handled, the Data Distribution Service will attempt to handle the DATA_AVAILABLE status and try to activate the DataReaderListener. In case this Listener is not activated for this status the event will follow the propagation rules as described above.

Conditions and Waitsets

The `Conditions` in conjunction with `WaitSets` provide for a state-based mechanism to synchronously inform the application of status changes. A `Condition` can be either a `ReadCondition`, `QueryCondition`, `StatusCondition` or `GuardCondition`. To create a `Condition` one of the following operations can be used:

- `ReadCondition` created by `create_readcondition`;

- `QueryCondition` created by `create_querycondition`;

- `StatusCondition` retrieved by `get_statuscondition` on an `Entity`;

- `GuardCondition` created by the Java operation `new`.

- Note that the `QueryCondition` is a specialized `ReadCondition`. The `GuardCondition` is a different kind of `Condition` since it is not controlled by a status but directly by the application (when a `GuardCondition` is initially created, the `trigger_value` is `false`). The `StatusCondition` is present by default with each `Entity`, therefore, it does not have to be created.
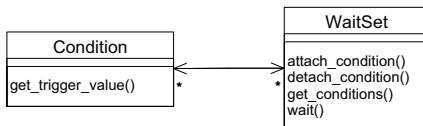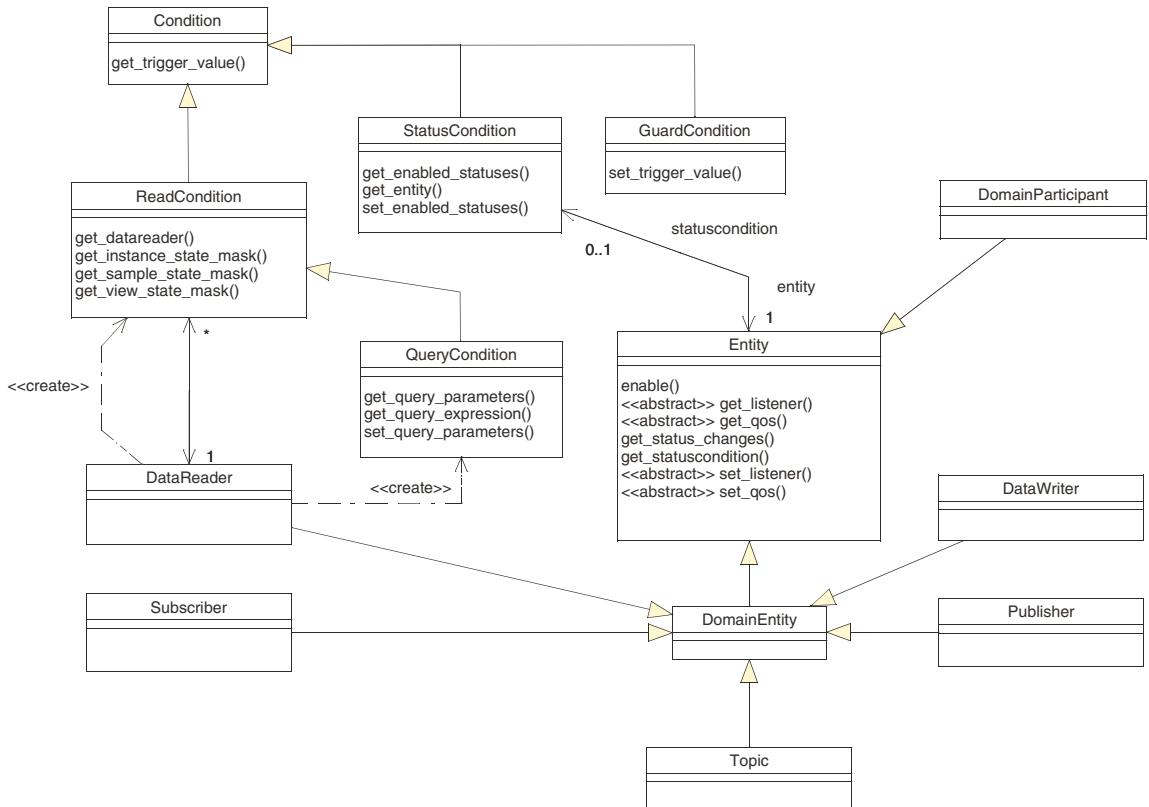
Figure 28 DCPS WaitSets

A `WaitSet` may have one or several `Conditions` attached to it. An application thread may block execution (blocking may be limited by a timeout) by waiting on a `WaitSet` until the `trigger_value` of one or more of the `Conditions` become `true`. When a `Condition`, whose `trigger_value` evaluates to `true`, is attached to a `WaitSet` that is currently being waited on (using the `wait` operation), the `WaitSet` will unblock immediately.

This (state-based) mechanism is generally used as follows:

- The application creates a `WaitSet`.

- The application indicates which relevant information it wants to be notified of, by creating or retrieving `Condition` objects (`StatusCondition`, `ReadCondition`, `QueryCondition` or `GuardCondition`) and attach them to a `WaitSet`.

- It then waits on that `WaitSet` (using `WaitSet.wait`) until the `trigger_value` of one or several `Condition` objects (in the `WaitSet`) become `true`.

- When the thread is unblocked, the application uses the result of the `wait` (i.e., the list of `Condition` objects with `trigger_value==true`) to actually get the information:

- if the condition is a `StatusCondition` and the status changes refer to a plain communication status, by calling `get_status_changes` and then `get_<communication_status>` on the relevant `Entity`;

- if the condition is a `StatusCondition` and the status changes refer to the read communication status:

- `DATA_ON_READERS`, by calling `get_status_changes` and then `get_datareaders` on the relevant `Subscriber` and then `read/take` on the returned `DataReader` objects;

- `DATA_AVAILABLE`, by calling `get_status_changes` and then `read/take` on the relevant `DataReader`.

- if it is a `ReadCondition` or a `QueryCondition`, by calling directly `read_w_condition/take_w_condition` on the `DataReader` with the `Condition` as a parameter.

Figure 29 DCPS Conditions

No extra information is passed from the Data Distribution Service to the application when a `wait` returns only the list of triggered `Condition` objects. Therefore, it is the application responsibility to investigate which `Condition` objects have triggered the `WaitSet`.

Blocking Behaviour

The result of a `wait` operation depends on the state of the `WaitSet`, which in turn depends on whether at least one attached `Condition` has a `trigger_value` of `true`. If the `wait` operation is called on `WaitSet` with state BLOCKED it will block the calling thread. If `wait` is called on a `WaitSet` with state UNBLOCKED it will return immediately. In addition, when the `WaitSet` transitions from state BLOCKED to state UNBLOCKED it wakes up the thread (if any) that had called `wait` on it. Note that there can only be one thread waiting on a single `WaitSet`.
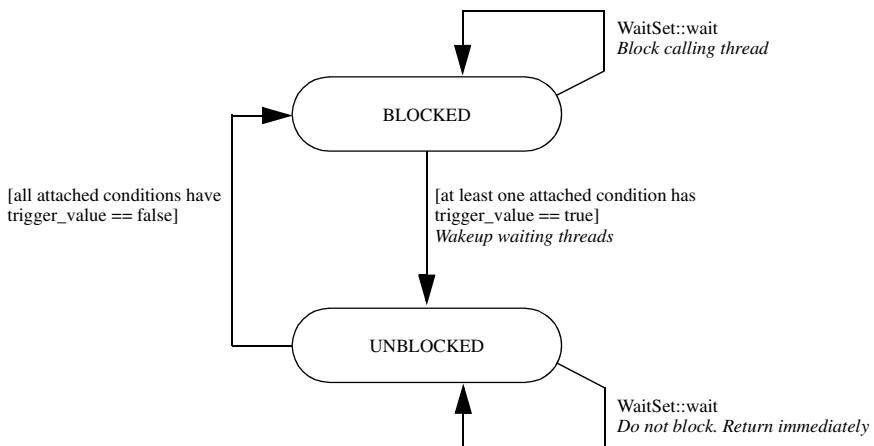
Figure 30 Blocking Behaviour of a Waitset State Chart

StatusCondition Trigger State

The `trigger_value` of a `StatusCondition` is the boolean OR of the `StatusChangedFlag` of all the communication statuses to which it is sensitive. That is, `trigger_value==false` only if all the values of the `StatusChangedFlags` are `false`.

The sensitivity of the `StatusCondition` to a particular communication status is controlled by the bit mask of `enabled_statuses` set on the `Condition` by means of the `set_enabled_statuses` operation.

ReadCondition and QueryCondition Trigger State

Similar to the `StatusCondition`, a `ReadCondition` also has a `trigger_value` that determines whether the attached `WaitSet` is BLOCKED or UNBLOCKED. However, unlike the `StatusCondition`, the `trigger_value` of the `ReadCondition` is tied to the presence of at least one sample managed by the Data Distribution Service with `SampleState`, `ViewState`, and `InstanceState` matching those of the `ReadCondition`. Additionally, for the `QueryCondition`, the data associated with the sample, must be such that the `query_expression` evaluates to `true`.

The fact that the trigger_value of a `ReadCondition` is dependent on the presence of samples on the associated `DataReader` implies that a single `take` operation can potentially change the `trigger_value` of several `ReadCondition` or `QueryCondition` objects.

For example, if all samples are taken, any `ReadCondition` or `QueryCondition` objects associated with the `DataReader` that had their `trigger_value==true` before will see the `trigger_value` change to `false`. Note that this does not guarantee that `WaitSet` objects, that had those `Condition` objects separately attached to, will not be woken up. Once we have `trigger_value==true` on a `Condition` it may wake up the `WaitSet` it was attached to, the condition transitions to `trigger_value==false` does not 'un-wake up' the `WaitSet` as 'un-wakening' is not possible. The consequence is that an application blocked on a `WaitSet` may return from the wait with a list of `Condition` objects some of which are no longer "active".

This is unavoidable if multiple threads are concurrently waiting on separate `WaitSet` objects and taking data associated with the same `DataReader Entity`. In other words, a `wait` may return with a list of `Condition` objects which all have a `trigger_value==false`. This only means that at some point one or more of the `Condition` objects have had a `trigger_value==true` but no longer do.

GuardCondition Trigger State

The `trigger_value` of a `GuardCondition` is completely controlled by the application via the operation `set_trigger_value`. This `Condition` can be used to implement an application defined wake-up of the blocked thread.

Appendix

G *Topic Definitions*

The Data Distribution Service distributes its data in structured data types, called topics. The first step when using the Data Distribution Service consists of defining these topics. Since the Data Distribution Service supports using several programming languages, OMG IDL is used for this purpose. This appendix describes how to define the topics.

Topic Definition Example

All data distributed using the Data Distribution Service has to be defined as a topic. A topic is a structured data type, like a class with several data members. Whenever the application needs to read or write data, it will be reading or writing topics. The definition of each topic it will be using has to be written in (a subset of) OMG IDL. For example:

```
module SPACE {
    struct Foo {
        long      userID;  // owner of message
        long long index;   // message index per owner
        string    content; // message body
    };
#pragma keylist Foo
    };
```

This is the definition of a topic called Foo, used for sending and receiving messages (as an example). Even though the topic is defined using IDL, the Data Distribution Service will be using an equivalent Java object which is accessed by the application using the type specific operations. Generation of the typed classes is achieved by invoking the Data Distribution Service IDL preprocessor, *idlpp -l java -S <idl_filename>.idl*, a tool which translates the IDL topic definition into an equivalent Java-definition. The *-l* java option indicates that Java code has to be generated. The -S option indicates that this Java-code should be StandAlone Java-code, i.e. it must not have any dependency on external ORB libraries. (It is also possible to use libraries from an existing ORB, so that your DDS application can also manage information coming from an external ORB. In that case you should use the CORBA-cohabitation mode, by replacing the -S flag witg a -C flag.) In this example, the pre-processor will generate the classes FooTypeSupport, FooDataWriter and FooDataReader which contain the type specific operations.

Complex Topics

The Foo topic is relatively simple, but the Data Distribution Service is capable of distributing more complex topics as well. In fact, any definition following the OpenSplice IDL subset is allowed. It is important to know that the pre-processor accepts all IDL constructs but only the subset is being processed.

Apart from the trivial data types, the Data Distribution Service is capable of handling fixed-length arrays, bounded and unbounded sequences, union types and enumerations. Types can be nested, e.g. a struct can contain a struct field or an array of structs, or a sequence of strings or an array of sequences containing structs.

IDL Preprocessor

The subset of OMG IDL that can be used to define the topics are described below.

IDL to Host Language Mapping

The Data Distribution Service IDL pre-processor translates the IDL-definition of the topics into language specific code. This translation is executed according to the OMG IDL mappings. Since the Data Distribution Service uses data-structures only, not all IDL-features are implemented by the pre-processor. Usually, the IDL definition consists of a module defining several structs and typedefs.

Data Distribution Service IDL Keywords

The identifiers listed in this appendix are reserved for use as keywords in IDL and may not be used otherwise, unless escaped with a leading underscore.

abstract	exception	inout	provides	truncatable
any	emits	interface	public	typedef
attribute	enum	local	publishes	typeid
boolean	eventtype	long	raises	typeprefix
case	factory	module	readonly	unsigned
char	false	multiple	setraises	union
component	finder	native	sequence	uses
const	fixed	Object	short	ValueBase
consumes	float	octet	string	valuetype
context	getraises	oneway	struct	void
custom	home	out	supports	wchar
default	import	primarykey	switch	wstring
double	in	private	true	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords are illegal. For example, `boolean` is a valid keyword; `Boolean` and `BOOLEAN` are illegal identifiers.

Data Distribution Service IDL Pragma Keylist

To define a topic, the content must either be a struct or a union. The pre-processor will only generate the type specific classes when topic definition is accompanied by a `<pragmakeylist>`. When the `<pragmakeylist>` has no `<field_id>`, the topic is available but no key is set. To define the keylist the definition, written in BNF-notation, is as follows:

```
<pragmakeylist>::= "#pragma keylist" <type_id> <field_id>*
<type_id>::= <struct_type_identifier>
       | <union_type_identifier>
<field_id>::= <member_declarator>
       | <element_spec_declarator>
```

In case of a struct, `<type_id>` is a `<struct_type_identifier>`. In case of a union, `<type_id>` is a `<union_type_identifier>`. The `<struct_type_identifier>` is the identifier used in the struct declaration. The `<union_type_identifier>` is the identifier used in the union declaration. The `<field_id>` is the identifier of a field in the struct or union identified by `<type_id>`. In case of a struct, `<field_id>` is a `<member_declarator>` which is one of the declarators used in the struct member. In case of a union, `<field_id>` is a `<element_spec_declarator>` which is one of the declarators used in the element specification in a case of the union.

For example, for the `Foo` example the next pragma must be used to have the pre-processor generate the typed classes (`FooTypeSupport`, `FooDataWriter` and `FooDataReader`).

```
#pragma keylist Foo userID index
```

Note that in this example the `userID` and the `index` are used as a key.

Data Distribution Service IDL Subset in BNF-notation

Only a subset of IDL is used by the pre-processor. A description of the Data Distribution Service IDL subset, written in BNF-notation, is as follows:

```
<definition> ::= <type_dcl> ";"
      | <const_dcl> ";"
      | <module> ";"
<module>::= "module" <identifier> "{" <definition>+ "}"
<scoped_name>::= <identifier>
      | "::" <identifier>
      | <scoped_name> "::" <identifier>
<const_dcl>::= "const" <const_type>
        <identifier> "=" <const_exp>
<const_type>::= <integer_type>
      | <char_type>
```

```
            |  <boolean_type>
            |  <floating_pt_type>
            |  <string_type>
            |  <scoped_name>
            |  <octet_type>
<const_exp>::= <or_expr>
<or_expr>::= <xor_expr>
            |  <or_expr> "|" <xor_expr>
<xor_expr>::= <and_expr>
            |  <xor_expr> "^" <and_expr>
<and_expr>::= <shift_expr>
            |  <and_expr> "&" <shift_expr>
<shift_expr>::= <add_expr>
            |  <shift_expr> ">>" <add_expr>
            |  <shift_expr> "<<" <add_expr>
<add_expr>::= <mult_expr>
            |  <add_expr> "+" <mult_expr>
            |  <add_expr> "-" <mult_expr>
<mult_expr>::= <unary_expr>
            |  <mult_expr> "*" <unary_expr>
            |  <mult_expr> "/" <unary_expr>
            |  <mult_expr> "%" <unary_expr>
<unary_expr>::= <unary_operator> <primary_expr>
            |  <primary_expr>
<unary_operator>::= "-"
            |  "+"
            |  "~"
<primary_expr>::= <scoped_name>
            |  <literal>
            |  "(" <const_exp> ")"
<literal>::= <integer_literal>
            |  <string_literal>
            |  <character_literal>
            |  <floating_pt_literal>
            |  <boolean_literal>
<boolean_literal>::= "TRUE"
            |  "FALSE"
<positive_int_const>::= <const_exp>
<type_dcl>::= "typedef" <type_declarator>
            |  <struct_type>
            |  <union_type>
            |  <enum_type>
<type_declarator>::= <type_spec> <declarators>
<type_spec>::= <simple_type_spec>
            |  <constr_type_spec>
<simple_type_spec>::= <base_type_spec>
            |  <template_type_spec>
            |  <scoped_name>
<base_type_spec>::= <floating_pt_type>
            |  <integer_type>
```

```
        | <char_type>
        | <boolean_type>
        | <octet_type>
<template_type_spec>::= <sequence_type>
        | <string_type>
<constr_type_spec>::= <struct_type>
        | <union_type>
        | <enum_type>
<declarators>::= <declarator> { "," <declarator> }*
<declarator>::= <simple_declarator>
        | <complex_declarator>
<simple_declarator>::= <identifier>
<complex_declarator>::= <array_declarator>
<floating_pt_type>::= "float"
        | "double"
<integer_type>::= <signed_int>
        | <unsigned_int>
<signed_int>::= <signed_short_int>
        | <signed_long_int>
        | <signed_longlong_int>
<signed_short_int>::= "short"
<signed_long_int>::= "long"
<signed_longlong_int>::= "long" "long"
<unsigned_int>::= <unsigned_short_int>
        | <unsigned_long_int>
        | <unsigned_longlong_int>
<unsigned_short_int>::= "unsigned" "short"
<unsigned_long_int>::= "unsigned" "long"
<unsigned_longlong_int>::= "unsigned" "long" "long"
<char_type>::= "char"
<boolean_type>::= "boolean"
<octet_type>::= "octet"
<struct_type>::= "struct" <identifier> "{" <member_list> "}"
<member_list>::= <member>+
<member>::= <type_spec> <declarators> ";"
<union_type>::= "union" <identifier> "switch"
        "(" <switch_type_spec> ")"
        "{" <switch_body> "}"
<switch_type_spec>::= <integer_type>
        | <char_type>
        | <boolean_type>
        | <enum_type>
        | <scoped_name>
<switch_body>::= <case>+
<case>::= <case_label>+ <element_spec> ";"
<case_label>::= "case" <const_exp> ":"
        | "default" ":"
<element_spec>::= <type_spec> <declarator>
<enum_type>::= "enum" <identifier>
        "{" <enumerator> { "," <enumerator> }* "}"
```

```
<enumerator>::= <identifier>
<sequence_type>::= "sequence" "<" <simple_type_spec> ","
      <positive_int_const> ">"
    | "sequence" "<" <simple_type_spec> ">"
<string_type>::= "string" "<" <positive_int_const> ">"
    | "string"
<array_declarator>::= <identifier> <fixed_array_size>+
<fixed_array_size>::= "[" <positive_int_const> "]"
```

PRISMTECH

H *DCPS Queries and Filters*

A subset of SQL syntax is used in several parts of OpenSplice:

- the `filter_expression` in the `ContentFilteredTopic`
- the `topic_expression` in the `MultiTopic`
- the `query_expression` in the `QueryReadCondition`

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:

- the `NonTerminals` are typeset in italics
- the `'Terminals'` are quoted and typeset in a fixed width font
- the `TOKENS` are typeset in small caps
- the notation *(element // ',')* represents a non-empty comma-separated list of *elements*

SQL Grammar in BNF

```
Expression::= FilterExpression
      | TopicExpression
      | QueryExpression
FilterExpression::= Condition
TopicExpression::= SelectFrom {Where } ';'
QueryExpression::= {Condition}
SelectFrom::= 'SELECT' Aggregation 'FROM' Selection
Aggregation::= '*'
      | (SubjectFieldSpec // ',')
SubjectFieldSpec::= FIELDNAME
      | FIELDNAME 'AS' FIELDNAME
      | FIELDNAME FIELDNAME
Selection::= TOPICNAME
      | TOPICNAME NaturalJoin JoinItem
JoinItem::= TOPICNAME
      | TOPICNAME NaturalJoin JoinItem
      | '(' TOPICNAME NaturalJoin JoinItem ')'
NaturalJoin::= '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::= FIELDNAME RelOp Parameter
      | Parameter RelOp FIELDNAME
BetweenPredicate::= FIELDNAME 'BETWEEN' Range
      | FIELDNAME 'NOT BETWEEN' Range
RelOp::= '=' | '>' | '>=' | '<' | '<=' | '<>' | like
Range::= Parameter 'AND' Parameter
Parameter::= INTEGERVALUE
        | FLOATVALUE
        | STRING
        | ENUMERATEDVALUE
        | PARAMETER
```

Note: INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. The aliases are all supported because they all are part of the SQL standard.

SQL Token Expression

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

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 field-name 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 Java mapping of the structure

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

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

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 post-fixed, which has the syntax en, where n is a number, optionally preceded by a plus or minus sign

STRING - Any series of characters encapsulated in single quotes, except a new-line character or a right quote. A string starts with a left or right quote, but ends with a right quote

ENUMERATEDVALUE - An enumerated value is a reference to a value declared within an enumeration. The name of the value must correspond to the names specified in the IDL definition of the enumeration, and must be encapsulated in single quotes. An enum value starts with a left or right quote, but ends with a right quote

PARAMETER - A parameter is of the form %*n*, where *n* represents a natural number (zero included) smaller than 100. It refers to the $n + 1$th argument in the given context

Note: when RelOp is 'like', Unix filename wildcards must be used for strings instead of the normal SQL wildcards. This means any one character is '?', any zero or more characters is '*'.

SQL Examples

Assuming Topic "Location" has as an associated type a structure with fields "flight_name, x, y, z", and Topic "FlightPlan" has as fields "flight_id, source, destination". The following are examples of using these expressions.

Example 1 topic_expression

```
"SELECT flight_name, x, y, z AS height FROM 'Location'
NATURAL JOIN 'FlightPlan' WHERE height < 1000 AND x <23"
```

Example 2 query_expression or a filter_expression

```
"height < 1000 AND x <23"
```

I

Built-in Topics

As part of its operation, the middleware must discover and possibly keep track of the presence of remote entities such as a new participant 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.

To make this information accessible to the application, the DCPS specification introduces a set of built-in topics and corresponding DataReader objects that can then be used by the application. The information is then accessed as normal application data. This approach avoids introducing a new API to access this information and allows the application to become aware of any changes in those values by means of any of the mechanisms presented in Appendix F, *Listeners, Conditions and Waitsets*.

The built-in data-readers all belong to a built-in Subscriber. This subscriber can be retrieved by using the method `get_builtin_subscriber` provided by the DomainParticipant (for details, see Section 3.2.1.16, *get_builtin_subscriber*, on page 164). The built-in DataReader objects can be retrieved by using the operation `lookup_datareader`, with the Subscriber and the topic name as parameter (for details, see Section 3.5.1.15, *lookup_datareader*, on page 359).

The QoS of the built-in Subscriber and DataReader objects is given by the following table:

Table 25 built-in Subscriber and DataReader QoS

USER_DATA	`<empty>`
TOPIC_DATA	`<empty>`
GROUP_DATA	`<empty>`
DURABILITY	TRANSIENT
DURABILITY_SERVICE	service_cleanup_delay = 0 history_kind = KEEP_LAST history_depth = 1 max_samples = LENGTH_UNLIMITED max_instances = LENGTH_UNLIMITED max_samples_per_instance = LENGTH_UNLIMITED

Table 25 built-in Subscriber and DataReader QoS (continued)

PRESENTATION	access_scope = TOPIC coherent_access = false ordered_access = false
DEADLINE	Period = INFINITE
LATENCY_BUDGET	duration = 0
OWNERSHIP	SHARED
LIVELINESS	kind = AUTOMATIC lease_duration = 0
TIME_BASED_FILTER	minimum_separation = 0
PARTITION	__BUILT-IN PARTITION__
RELIABILITY	kind = RELIABLE max_blocking_time = 100 milliseconds synchronous = false
DESTINATION_ORDER	BY_RECEPTION_TIMESTAMP
HISTORY	kind = KEEP_LAST depth = 1
RESOURCE_LIMITS	max_samples = LENGTH_UNLIMITED max_instances = LENGTH_UNLIMITED max_samples_per_instance = LENGTH_UNLIMITED
READER_DATA_LIFECYCLE	autopurge_nowriter_samples_delay = INFINITE autopurge_disposed_samples_delay = INFINITE invalid_sample_visibility = MINIMUM_INVALID_SAMPLES
ENTITY_FACTORY	autoenable_created_entities = true
SHARE	enable = false name = null
READER_DATA_LIFESPAN	used = false duration = INFINITE
USER_KEY	enable = false expression = null

Built-in entities have default listener settings as well. The built-in Subscriber and all
of its built-in Topics have nil listeners with all statuses appearing in their listener
masks. The built-in DataReaders have nil 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. The QoS policies appear as normal `''` 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 tables below list the built-in topics, their names, and the additional information (beyond the QoS policies that apply to the remote entity) that appears in the data associated with the built-in topic.

ParticipantBuiltinTopicData

The `DCPSParticipant` topic communicates the existence of DomainParticipants by means of the `ParticipantBuiltinTopicData` datatype. Each `ParticipantBuiltinTopicData` sample in a Domain represents a DomainParticipant that participates in that Domain: a new `ParticipantBuiltinTopicData` instance is created when a newly added DomainParticipant is enabled, and it is disposed when that DomainParticipant is deleted. An updated `ParticipantBuiltinTopicData` sample is written each time the DomainParticipant modifies its `UserDataQosPolicy`.

Table 26 ParticipantBuiltinTopicData Members

Name	Type	Description
key	BuiltinTopicKey_t	Globally unique identifier of the participant
user_data	UserDataQosPolicy	User-defined data attached to the participant via a QosPolicy

TopicBuiltinTopicData

The `DCPSTopic` topic communicates the existence of topics by means of the `TopicBuiltinTopicData` datatype. Each `TopicBuiltinTopicData` sample in a Domain represents a Topic in that Domain: a new `TopicBuiltinTopicData` instance is created when a newly added Topic is enabled. However, the instance is not disposed when a Topic is deleted by its participant because a topic lifecycle is tied to the lifecycle of a Domain, not to the lifecycle of an individual participant. (See also Section 3.2.1.13, *delete_topic*, on page 161, which explains that a DomainParticipant can only delete its local proxy to the real Topic). An updated `TopicBuiltinTopicData` sample is written each time a Topic modifies one or more of its QosPolicy values.

Information published in the *DCPSTopicTopic* is critical to the data distribution service, therefore it cannot be disabled by means of the Domain/BuiltinTopics element in the configuration file.

Table 27 TopicBuiltinTopicData Members

Name	Type	Description
key	BuiltinTopicKey_t	Global unique identifier of the Topic
name	String	Name of the Topic
type_name	String	Type name of the Topic (i.e. the fully scoped IDL name)
durability	DurabilityQosPolicy	QosPolicy attached to the Topic
durability_service	DurabilityServiceQosPolicy	QosPolicy attached to the Topic
deadline	DeadlineQosPolicy	QosPolicy attached to the Topic
latency_budget	LatencyBudgetQosPolicy	QosPolicy attached to the Topic
liveliness	LivelinessQosPolicy	QosPolicy attached to the Topic
reliability	ReliabilityQosPolicy	QosPolicy attached to the Topic
transport_priority	TransportPriorityQosPolicy	QosPolicy attached to the Topic
lifespan	LifespanQosPolicy	QosPolicy attached to the Topic
destination_order	DestinationOrderQosPolicy	QosPolicy attached to the Topic
history	HistoryQosPolicy	QosPolicy attached to the Topic
resource_limits	ResourceLimitsQosPolicy	QosPolicy attached to the Topic
ownership	OwnershipQosPolicy	QosPolicy attached to the Topic
topic_data	TopicDataQosPolicy	QosPolicy attached to the Topic

PublicationBuiltinTopicData

The *DCPSPublication* topic communicates the existence of datawriters by means of the `PublicationBuiltinTopicData` datatype. Each `PublicationBuiltinTopicData` sample in a Domain represents a datawriter in that Domain: a new `PublicationBuiltinTopicData` instance is created when a newly added DataWriter is enabled, and it is disposed when that DataWriter is deleted. An updated `PublicationBuiltinTopicData` sample is written each time the DataWriter (or the Publisher to which it belongs) modifies a QosPolicy that applies to the entities connected to it. Also will it be updated when the writer looses or regains its liveliness.

The `PublicationBuiltinTopicData` Topic is also used to return data through the `get_matched_publication_data` operation on the DataReader.

Table 28 PublicationBuiltinTopicData Members

Name	Type	Description
`key`	BuiltinTopicKey_t	Global unique identifier of the DataWriter
`participant_key`	BuiltinTopicKey_t	Global unique identifier of the Participant to which the DataWriter belongs
`topic_name`	String	Name of the Topic used by the DataWriter
`type_name`	String	Type name of the Topic used by the DataWriter
`durability`	DurabilityQosPolicy	QosPolicy attached to the DataWriter
`deadline`	DeadlineQosPolicy	QosPolicy attached to the DataWriter
`latency_budget`	LatencyBudgetQosPolicy	QosPolicy attached to the DataWriter
`liveliness`	LivelinessQosPolicy	QosPolicy attached to the DataWriter
`reliability`	ReliabilityQosPolicy	QosPolicy attached to the DataWriter
`lifespan`	LifespanQosPolicy	QosPolicy attached to the DataWriter
`destination_order`	DestinationOrderQosPolicy	QosPolicy attached to the DataWriter
`user_data`	UserDataQosPolicy	QosPolicy attached to the DataWriter
`ownership`	OwnershipQosPolicy	QosPolicy attached to the DataWriter
`ownership_strength`	OwnershipStrengthQosPolicy	QosPolicy attached to the DataWriter
`presentation`	PresentationQosPolicy	QosPolicy attached to the Publisher to which the DataWriter belongs
`partition`	PartitionQosPolicy	QosPolicy attached to the Publisher to which the DataWriter belongs
`topic_data`	TopicDataQosPolicy	QosPolicy attached to the Topic used by the DataWriter
`group_data`	GroupDataQosPolicy	QosPolicy attached to the Publisher to which the DataWriter belongs

SubscriptionBuiltinTopicData

The `DCPSSubscription` topic communicates the existence of datareaders by means of the `SubscriptionBuiltinTopicData` datatype. Each `SubscriptionBuiltinTopicData` sample in a Domain represents a datareader in that Domain: a new `SubscriptionBuiltinTopicData` instance is created when a newly added DataReader is enabled, and it is disposed when that

DataReader is deleted. An updated `SubscriptionBuiltinTopicData` sample is written each time the DataReader (or the Subscriber to which it belongs) modifies a QosPolicy that applies to the entities connected to it.

Table 29 SubscriptionBuiltinTopicData Members

Name	Type	Description
`key`	BuiltinTopicKey_t	Global unique identifier of the DataReader
`participant_key`	BuiltinTopicKey_t	Global unique identifier of the Participant to which the DataReader belongs
`topic_name`	String	Name of the Topic used by the DataReader
`type_name`	String	Type name of the Topic used by the DataReader
`durability`	DurabilityQosPolicy	QosPolicy attached to the DataReader
`deadline`	DeadlineQosPolicy	QosPolicy attached to the DataReader
`latency_budget`	LatencyBudgetQosPolicy	QosPolicy attached to the DataReader
`liveliness`	LivelinessQosPolicy	QosPolicy attached to the DataReader
`reliability`	ReliabilityQosPolicy	QosPolicy attached to the DataReader
`ownership`	LifespanQosPolicy	QosPolicy attached to the DataReader
`destination_order`	DestinationOrderQosPolicy	QosPolicy attached to the DataReader
`user_data`	UserDataQosPolicy	QosPolicy attached to the DataReader
`time_based_filter`	TimeBasedFilterQosPolicy	QosPolicy attached to the DataReader
`presentation`	PresentationQosPolicy	QosPolicy attached to the Subscriber to which the DataReader belongs
`partition`	PartitionQosPolicy	QosPolicy attached to the Subscriber to which the DataReader belongs
`topic_data`	TopicDataQosPolicy	QosPolicy attached to the Topic used by the DataReader
`group_data`	GroupDataQosPolicy	QosPolicy attached to the Subscriber to which the DataReader belongs

Other builtin topics

There are a number of other built-in topics that have not been mentioned. These topics (*e.g.* `DCPSDelivery`, `DCPSHeartbeat` and potentially some others) are proprietary and for internal use only. Users are discouraged from doing anything with these topics, so as not to interfere with internal mechanisms that rely on them. The structure of these topics may change without notification.

PRISMTECH

BIBLIOGRAPHY

Bibliography

[1] *OMG Data Distribution Service Revised Final Adopted Specification ptc/04-03-07*, Object Management Group

[2] *OMG Java Language Mapping Specification formal/02-08-05*, Object Management Group (OMG)

[3] *OMG The Common Object Request Broker: Architecture and Specification*, Version 3.0, formal/02-06-01, Object Management Group

GLOSSARY

Glossary

Acronyms

Acronym	Meaning
CORBA	Common Object Request Broker Architecture
DCPS	Data Centric Publish/Subscribe
DDS	Data Distribution Service
IDL	Interface Definition Language
OMG	Object Management Group
ORB	Object Request Broker
QoS	Quality of Service
SPLICE	Subscription Paradigm for the Logical Interconnection of Concurrent Engines

Glossary

INDEX

Index

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W