PRISMTECH
AN ADLINK COMPANY

Streams API Reference Guide

Release 6.x

Contents

1

Preface

1.1 About The Vortex Streams API Reference Guide

The *Vortex Streams API Reference Guide* provides a detailed overview of the Vortex OpenSplice Streams API. The Streams API is an add-on, built on the Data Centric Public Subscribe (DCPS) paradigm that is implemented by Vortex OpenSplice and standardized in the OMG's Data Distribution Service Specification.

This Guide complements the Vortex OpenSplice *C++ Reference Guide*.

1.2 Intended Audience

The *Streams API Reference Guide* is intended to be used by C++ programmers who are using the OpenSplice Streams API to develop applications. While not strictly required, it is assumed that the reader has a basic understanding of the DDS C++ API as detailed in the Vortex OpenSplice *C++ Reference Guide*.

1.3 Organisation

This *Guide* is organised in two parts.

The *Introduction* provides some background information about the features of the Streams API and how to use them. It also gives a broad overview of all entities and relations between entities in the Streams API.

The *API Reference* provides detailed descriptions of all of the classes and operations of the Streams API.

1.4 Conventions

The icons shown below are used in PrismTech product documentation to help readers to quickly identify information relevant to their specific use of Vortex OpenSplice.

Icon	*Meaning*
⚠	Item of special significance or where caution needs to be taken.
ℹ	Item contains helpful hint or special information.
Windows	Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.
Unix	Information applies to Unix-based systems (*e.g.* Solaris) only.
Linux	Information applies to Linux-based systems (*e.g.* Ubuntu) only.
C	C language specific.
C++	C++ language specific.
C#	C# language specific.
Java	Java language specific.

2

Introduction

2.1 Features

Vortex OpenSplice Streams API supports a common data-distribution pattern where continuous flows or *streams* of data have to be transported with minimal overhead and therefore maximal achievable throughput.

Vortex OpenSplice Streams API implements this *streams pattern* by transparent packing and queuing of data samples using auto-generated *containers*, thus minimizing the overhead normally associated with the management and distribution of individual DDS samples.

2.2 Getting Started

The Vortex OpenSplice Streams API is divided in two main components:

- type-specific code that can be generated using the OpenSplice IDL Pre-Processor
- a Streams library.

Applications that wish to use the Streams API are required to do two things:

1. Link against *one* of the OpenSplice Streams libraries available within the Vortex OpenSplice distribution. There are separate libraries for either *CORBA-Cohabitation* mode or *Standalone C++* mode.
2. Annotate the data-model IDL file with `#pragma` stream directives for each data type for which a Stream needs to be created.

The Vortex OpenSplice Streams API is built on the DCPS API. Since the C++ bindings of Vortex OpenSplice are available in two flavours, so is the Streams API. In the following paragraphs the steps will be discussed to build a simple application that uses the following data-model:

```
Space.idl:

module Space {
    struct Foo {
        long long_1;
        long long_2;
    };
#pragma stream Foo

    struct Type2 {
      long long_1;
      long long_2;
      long long_3;
    };
#pragma stream Type2
#pragma keylist Type2 long_1

};
```

Using this model, both Foo and Type2 can be used with the Streams API. In addition Type2 can also be used as a regular DDS topic, with long_1 as key.

The following relevant Streams API classes are generated based on this model for Foo:

```
Space::FooStreamDataWriter
Space::FooStreamDataReader
Space::FooStreamBuf
```

It is recommended to use smart references to the StreamDataWriter and StreamDataReader classes in applications. The regular Vortex OpenSplice C++ smart-pointer <class>_var types are available for this purpose. See the section on *Memory Management* in the Vortex OpenSplice DDS *C++ Reference Guide* for more information.

2.2.1 CORBA Cohabitation Mode

In *CORBA Co-habitation* mode, idlpp generates code that can be processed with any of the supported ORB compilers (OpenFusion TAO, Mico, *etc.*).

First idlpp is executed on the Space.idl file:

```
$ idlpp -I$OSPL_HOME/etc/idl -l cpp -C Space.idl
```

The standard Vortex OpenSplice DDS IDL directory is referenced as include-path, since it contains definitions of some basic data-types and interfaces that are required if DDS Topics are created for any of the types in the IDL file. The other parameters are used to put idlpp in C++ CORBA-Cohabitation mode.

As usual when DDS topics are created, the above command generates, among other files, a file called SpaceDcps.idl. The file SpaceStreams.idl is also generated.

To proceed, idlpp should be executed on the ExampleStreams.idl file:

```
$ idlpp -I$OSPL_HOME/etc/idl -l cpp -C SpaceStreams.idl
```

This creates the descriptions of the DCPS entities that are required to manage the DDS topics that will be used for the Streams types, just like with the original IDL file, in a file called SpaceStreamsDcps.idl.

Now all four IDL files should be processed with the appropriate (ORB-specific) CORBA IDL processor. After this step all code and header files are generated to start using the Streams API in application code.

2.2.2 Standalone Mode

In *Standalone C++* mode, the generated interfaces are *not* required to be processed by an IDL compiler. Instead, idlpp will use the cppgen code-generator that is part of the Vortex OpenSplice DDS distribution. idlpp will automatically call cppgen to process certain files; the user is only required to execute idlpp, first on the original IDL file:

```
$ idlpp -I$OSPL_HOME/etc/idl -l cpp -S Space.idl
```

This creates SpaceStreams.idl, which in turn also needs to be processed by idlpp:

```
$ idlpp -I$OSPL_HOME/etc/idl -l cpp -S -i SpaceStreams.idl
```

The -i parameter is required because normally no code is generated for interfaces (for DDS topics, only datatypes are generated). In the case of streams, interfaces should not be ignored.

3

API Reference

3.1 Introduction

As described in :ref:‘Getting Started <Getting Started>‘, the OpenSplice IDL preprocessor generates typed Streams API classes for each type that is annotated with a streams pragma.

As in the OpenSplice DDS *C++ Reference Guide*, the fictional type `Foo`, defined in module Space, is used as an example. When the `Foo` type is annotated with a pragma streams, `FooStreamDataWriter` and `FooStreamDataReader` classes will be generated.

This section describes the usage of all operations on these classes.

3.2 QoS Policies

StreamDataWriterQos	StreamDataReaderQos
`StreamFlushQosPolicy`	

StreamFlushQosPolicy	Type	Default value
`max_delay`	`DDS::Duration_t`	`DDS::DURATION_INFINITE`
`max_samples`	`long`	`0`

3.2.1 StreamDataWriterQos

StreamFlushQosPolicy

Scope

`DDS::Streams`

Synopsis

```
#include <streams_ccpp.h>

struct StreamFlushQosPolicy {
   Duration_t  max_delay;
   long max_samples;
};
```

Description

The `StreamFlushQosPolicy` can be used to set limits on the stream(s) of the `StreamDataWriter` it is applied to.

Attributes

`Duration_t max_delay` Time-based limit. The StreamDataWriter will automatically flush all of its streams each `max_delay` period.

⚠ *Note*: `max_delay` is not yet implemented. It is scheduled for a future release.

`long max_samples` Samples-per-stream based limit. The StreamDataWriter will automatically flush a stream when, after appending a sample, the number of samples in that stream equals `max_samples`.

Detailed Description

By setting the `StreamFlushQosPolicy`, the `StreamDataWriter` will automatically flush its stream(s) based on a particular limit. The attributes can be combined, for example a `max_delay` of *1* second and a `max_samples` of *100* will result in a flush at least each second or sooner if 100 samples are appended to a stream.

The `max_delay` limit applies to all streams in case a `StreamDataWriter` manages more than one stream. It is initialized when the first stream is created, and applied to all streams created after that.

In case of a manual flush (when the application calls the flush operation), the `max_samples` limit is reinitialized.

StreamDataReaderQos

Currently no QoS properties for a `StreamDataReader` have been identified, but the `StreamDataReaderQos` is defined in the API to maintain consistency with the `StreamDataWriter`; it is reserved for future use.

3.3 StreamDataWriter Class

3.3.1 Constructors

Scope

`Space::FooStreamDataWriter`

Synopsis

```
#include <SpaceStreamsApi.h>

ooStreamDataWriter(
   DDS::Publisher_ptr publisher,
   DDS::Streams::StreamDataWriterQos &sqos,
   const char* streamName);

FooStreamDataWriter(
   DDS::DomainId_t domainId,
      DDS::Streams::StreamDataWriterQos &sqos,
      const char* streamName);

FooStreamDataWriter(
   DDS::Streams::StreamDataWriterQos &sqos,
      const char* streamName);

FooStreamDataWriter(
   DDS::Publisher_ptr publisher,
      const char* streamName);

FooStreamDataWriter(
      DDS::DomainId_t domainId,
      const char* streamName);
```

Description

Multiple constructors are available to create a `FooStreamDataWriter`. Depending on which parameters are supplied by the application, one of the overloaded constructors will be selected to create a new instance of the `FooStreamDataWriter` class.

Parameters

`in DDS::Publisher_ptr publisher` A pointer to a pre-created DDS Publisher. This parameter is optional; if a publisher is not supplied the `FooStreamDataWriter` will create an internal publisher.

`in DDS::DomainId_t domainId` The id of the DDS domain to attach to. The `DDS::DOMAIN_ID_DEFAULT` macro can be used to connect to the default domain, which is also used if the parameter is omitted.

`in DDS::Streams::StreamDataWriterQos &sqos` The QoS settings that are applied to the `FooStreamDataWriter`.

`in const char* streamName` The system-wide unique name of the stream that is used to create a DDS (container-)topic for the stream(s) that are handled by the `FooStreamDataWriter`.

Exceptions

Constructors cannot return a value, therefore they throw exceptions when the object cannot be constructed. Besides exceptions, the regular OpenSplice error logging framework is used to report additional information when a constructor fails.

The constructors throw a `StreamsException` if an error occurs. The application may catch these exceptions to detect when creation of a StreamDataWriter doesn't succeed.

```
DDS::Streams::StreamsException {
   out const char *message;
   out DDS::ReturnCode_t id
}
```

The message contains a description of the error. The `id` field contains a DDS error code that represents the error condition.

Detailed Description

When a pre-created publisher is not supplied, the `FooStreamDataWriter` will create an internal DDS participant and DDS publisher. This will naturally consume some resources, so when a lot of streams need to be created it is recommended to supply a publisher that can be re-used for each `FooStreamDataWriter` instance.

The `streamName` is a required parameter. The `FooStreamDataWriter` will create a DDS topic of the correct type and name it after the supplied `streamName`.

3.3.2 append

Scope

`Space::FooStreamDataWriter`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
append(
   StreamId id,
   const Foo &data)
```

Description

Write a sample to the stream with the supplied `id`.

Parameters

in StreamId id The stream id.

in Foo &data The data to write to the stream.

Return Value

ReturnCode_t Possible return codes of the operation are: DDS::RETCODE_OK, DDS::RETCODE_PRECONDITION_NOT_MET.

Detailed Description

Using the append operation, the application can write data to a stream. Note that for each stream of a certain type, multiple *instances* of this stream-type can be created by assigning unique ids to each of streams. Each id then represents an *instance* of the stream of the associated type. So the actual stream instance is selected based on the supplied StreamId.

When the stream doesn't exist it is automatically created based on the current QoS settings.

Return Code

When the operation returns:

RETCODE_OK The data was successfully appended to the stream.

RETCODE_PRECONDITION_NOT_MET A precondition failed, data was not appended.

If the StreamDataWriter QoS specifies an auto-flush maximum samples limit, an append may trigger a flush. In that case the append call forwards the return code of the flush to the application, so any return code that is specified in the next section may also be returned by append.

3.3.3 flush

Scope

Space::FooStreamDataWriter

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
flush(
   DDS::Streams::StreamId id)
```

Description

Write all data in a stream to the DDS subsystem.

Parameters

in StreamId id The id of the stream.

Return Value

ReturnCode_t Possible return codes of the operation are: DDS::RETCODE_OK, DDS::RETCODE_PRECONDITION_NOT_MET.

Detailed Description

When a stream is flushed, all data in the stream is delivered to DDS and the stream is emptied. The memory allocated will be reused the next time data is appended to the stream.

The flush operation results in a write call on the underlying DDS subsystem. Depending on the result of the write, this result is returned back to the application.

Return Code

RETCODE_OK The stream was successfully flushed.

RETCODE_PRECONDITION_NOT_MET A precondition failed; most likely the stream doesn't exist.

See the OpenSplice DDS *C++ Reference Guide* for possible result codes returned by a DDS `write` operation.

3.3.4 get_qos

Scope

`Space::FooStreamDataWriter`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
get_qos(
   DDS::Streams::StreamDataWriterQos &qos)
```

Description

This operation allows access to the existing set of QoS policies for a `FooStreamDataWriter`.

Parameters

inout StreamDataWriterQos &qos A pointer to a `StreamDatatWriterQos` object to which the current QoS settings will be copied.

Return Value

ReturnCode_t Possible return code of the operation is: `DDS::RETCODE_OK`.

Detailed Description

The existing list of QoS settings of the `FooStreamDataWriter` is copied to the object pointed to by `qos`. The application can then inspect and, if necessary, modify the settings and apply the settings using the `set_qos` operation.

Return Code

RETCODE_OK The QoS settings were successfully copied to the supplied `qos` object.

3.3.5 set_qos

Scope

`Space::FooStreamDataWriter`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
set_qos(
   DDS::Streams::StreamDataWriterQos &qos)
```

Description

This operation allows replacing the existing set of QoS policies for a `FooStreamDataWriter`.

Parameters

in StreamDataWriterQos &qos A pointer to a `qos` object with the new policies.

Return Value

ReturnCode_t Possible return codes of the operation are: `DDS::RETCODE_OK`, `DDS::RETCODE_UNSUPPORTED`.

Detailed Description

This operation allows replacing the set of QoS policies of a `FooStreamDataWriter`.

⚠ *Note*: A new `StreamFlushQosPolicy` may decrease the value of `max_samples`, but existing streams are not allowed to violate this limit. Any streams that contain data that exceeds the new `max_samples` value are automatically flushed before the new policy is applied.

Return Code

RETCODE_OK The QoS settings were successfully applied to the `FooStreamDataWriter`.

RETCODE_UNSUPPORTED The application attempted to set QoS policies or values that are not (yet) supported.

3.4 StreamDataReader Class

3.4.1 Constructors

Scope

`Space::FooStreamDataReader`

Synopsis

```
#include <SpaceStreamsApi.h>

FooStreamDataReader(
   DDS::Subscriber_ptr subscriber,
   DDS::Streams::StreamDataReaderQos &sqos,
   const char* streamName);

FooStreamDataReader(
   DDS::DomainId_t domainId,
      DDS::Streams::StreamDataReaderQos &sqos,
      const char* streamName);

FooStreamDataReader(
   DDS::Streams::StreamDataReaderQos &sqos,
      const char* streamName);

FooStreamDataReader(
   DDS::Subscriber_ptr subscriber,
      const char* streamName);

FooStreamDataReader(
      DDS::DomainId_t domainId,
      const char* streamName);
```

Description

Multiple constructors are available to create a `FooStreamDataReader`. Depending on which parameters are supplied by the application, one of the overloaded constructors will be selected to create a new instance of a `FooStreamDataReader` class.

Parameters

in DDS::Subscriber_ptr subscriber A pointer to a pre-created DDS Subscriber. This parameter is optional; if a subscriber is not supplied the `FooStreamDataReader` will create an internal subscriber.

in DDS::DomainId_t domainId The id of the DDS domain to attach to. The DDS::DOMAIN_ID_DEFAULT macro can be used to connect to the default domain, which is also used if the parameter is omitted.

in DDS::Streams::StreamDataReaderQos &sqos The QoS settings that are applied to the FooStreamDataReader.

in const char* streamName The system-wide unique name of the stream which is also used to create a DDS (container-)topic for the stream(s) that are handled by the FooStreamDataReader.

Exceptions

Constructors cannot return a value, therefore they throw exceptions when the object cannot be constructed. Besides exceptions, the regular OpenSplice error logging framework is used to report additional information when a constructor fails.

The constructors throw a StreamsException if an error occurs. The application may catch these exceptions to detect when creation of a StreamDataReader doesn't succeed.

```
DDS::Streams::StreamsException {
   out const char *message;
   out DDS::ReturnCode_t id
}
```

The message contains a description of the error. The id field contains a DDS error code that represents the error condition.

Detailed Description

When a pre-created subscriber is not supplied, the FooStreamDataReader will create an internal DDS participant and DDS subscriber. This will naturally consume some resources, so when a lot of instances need to be created it is recommended to supply a subscriber that can be re-used for each FooStreamDataReader instance.

The streamName is a required parameter. The FooStreamDataReader will create a DDS topic of the correct type and name it after the supplied streamName.

3.4.2 get

Scope

Space::FooStreamDataReader

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
get(
   DDS::Streams::StreamId id,
   Space::FooStreamBuf data_values,
   long max_samples,
   DDS::Duration_t timeout);
```

Description

Check if any data is available in a stream and retrieve it, emptying the stream.

Parameters

in StreamId id The id of the stream instance from which to retrieve the data.

inout FooStreamBuf data_values The buffer in which the data is stored.

in long max_samples The maximum amount of data samples retrieved. Default is DDS::LENGTH_UNLIMITED.

in Duration_t timeout Blocking time, in case no data is immediately available.

Return Value

ReturnCode_t Possible return codes of the operation are: `DDS::RETCODE_OK`, `DDS::RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

Using the `get` operation, the application can retrieve data from a stream. The stream is selected based on the supplied `StreamId`.

If no data is available initially, the `get` operation blocks for a maximum period specified in the `timeout` parameter. If data becomes available during the `timeout` period the `FooStreamDataReader` proceeds to retrieve the data and return it to the application. To return immediately, the application can use the special value `DDS::DURATION_ZERO` as a `timeout` parameter. To block indefinitely until data is available, the value `DDS::DURATION_INFINITE` should be passed.

The data is returned in a buffer that is to be supplied by the application. The application is responsible for allocating a buffer that is large enough to contain the available data. If more data is available than will fit in the buffer, the excess data will be stored by the StreamDataReader and returned to the application during the next call to get (or get_w_filter). In this state, the `StreamDataReader` will only attempt to retrieve new data after all data that was stored internally is returned to the application.

Since allocating memory for the buffer is an expensive operation, it is recommended to re-use the same buffer for each subsequent call to get or get_w_filter. The `max_samples` parameter can be used to limit the amount of data that is returned with each get or get_w_filter call.

⚠ *Note*: Internal pre-allocation of buffers, using a loans registry similar to the DCPS API, will be implemented in a future version.

Return Code

DDS::RETCODE_OK Data is returned in the `data_values` buffer.

DDS::RETCODE_NO_DATA There is currently no data available.

DDS::RETCODE_PRECONDITION_NOT_MET The operation could not be performed because a precondition is not met; most likely the `data_values` buffer is not preallocated.

The list of possible return codes includes all possible return codes of `waitset.wait()` and `take_instance()` calls. These DCPS calls are used internally by the Streams API. There is one exception: if the `waitset.wait()` returns a `DDS::RETCODE_TIMEOUT`, this return code is translated to a `DDS::RETCODE_NO_DATA` return code.

See the OpenSplice DDS *C++ Reference Guide* for possible result codes returned by a DDS `take_instance` operation and for `waitset.wait()`.

3.4.3 get_w_filter

Scope

`Space::FooStreamDataReader`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
get_w_filter(
   DDS::Streams::StreamId id,
   Space::FooStreamBuf data_values,
   long max_samples,
   DDS::Duration_t timeout
   Space::FooStreamFilterCallback a_filter);
```

Description

Check if any data is available in a stream and retrieve it if it matches the filter, discard otherwise.

Parameters

in StreamId id The id of the stream instance of which to retrieve the data.

inout FooStreamBuf data_values The buffer in which the data is stored.

in long max_samples The maximum amount of data samples retrieved.

in Duration_t timeout Blocking time, in case no data is immediately available.

in FooStreamFilterCallback a_filter Pointer to a function that implements a filter for the data.

Return Value

ReturnCode_t Possible return codes of the operation are: DDS::RETCODE_OK, DDS::RETCODE_PRECONDITION_NOT_MET.

Detailed Description

The get_w_filter operation is equivalent to the get operation, the description of get also applies to get_w_filter.

The difference is that get_w_filter allows the application to supply a FooStreamFilterCallback instance that implements the match_data() operation. Each data sample is matched against the filter and only data for which the filter returns true is returned to the application.

Samples that do not match the filter are not considered in relation to max_samples and the data_values buffer length; the buffer does *not* need to be capable of holding *all* available samples, just the samples that pass the filter.

Samples are only evaluated once and are discarded if not matched.

Return Code

DDS::RETCODE_OK Data is returned in the data_values buffer.

DDS::RETCODE_NO_DATA There is no data available during the period specified by timeout.

DDS::RETCODE_PRECONDITION_NOT_MET The operation could not be performed because a precondition is not met; most likely the data_values buffer is not preallocated.

The list of possible return codes includes all possible return codes of waitset.wait() and take_instance() calls. These DCPS calls are used internally by the Streams API. There's one exception: If the waitset.wait() returns a DDS::RETCODE_TIMEOUT, this return code is translated to a DDS::RETCODE_NO_DATA return code.

See the OpenSplice DDS *C++ Reference Guide* for possible result codes returned by a DDS take_instance operation and waitset.wait().

3.4.4 return_loan

Scope

Space::FooStreamDataReader

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
return_loan(
   Space::FooStreamBuf data_values)
```

Description

The application should use this operation to indicate that it has finished accessing the sequence of `data_values`.

Parameters

`inout FooStreamBuf data_values` The data sequence which was loaned from the `FooStreamDataReader`.

Return Value

`ReturnCode_t` Possible return codes of the operation are: `DDS::RETCODE_OK`, `DDS::RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

When the application does not pre-allocate a buffer to hold the data, the `FooStreamDataReader` will do so itself when a get operation is invoked. The application calls `return_loan` to indicate that it has finished accessing this buffer so the `FooStreamDataReader` can reclaim the resources allocated for the buffer.

⚠ *Note*: Internal pre-allocation will be implemented in a future release. This operation has no effect on buffers allocated by the application.

3.4.5 get_qos

Scope

`Space::FooStreamDataReader`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
get_qos(
   DDS::Streams::StreamDataReaderQos &qos)
```

Description

This operation allows access to the existing set of QoS policies for a `FooStreamDataReader`.

Parameters

`inout StreamDataReaderQos &qos` A pointer to a `StreamDataReaderQos` object to which the current QoS settings will be copied.

Return Value

`ReturnCode_t` Possible return code of the operation is: `DDS::RETCODE_OK`.

Detailed Description

The existing list of QoS settings of the `FooStreamDataReader` is copied to the object pointed to by `qos`. The application can then inspect and, if necessary, modify the settings and apply the settings using the `set_qos` operation.

Return Code

`RETCODE_OK` The QoS settings were successfully copied to the supplied `qos` object.

3.4.6 set_qos

Scope

`Space::FooStreamDataReader`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
set_qos(
   DDS::Streams::StreamDataReaderQos &qos)
```

Description

This operation allows replacing the existing set of QoS policies for a `FooStreamDataReader`.

Parameters

`in StreamDataReaderQos &qos` A pointer to a `qos` object with the new policies.

Return Value

`ReturnCode_t` Possible return codes of the operation are: `DDS::RETCODE_OK`, `DDS::RETCODE_UNSUPPORTED`.

Detailed Description

This operation allows replacing the set of QoS policies of a `FooStreamDataReader`.

Return Code

`RETCODE_OK` The QoS settings were successfully applied to the `FooStreamDataWriter`.

`RETCODE_UNSUPPORTED` The application attempted to set QoS policies or values that are not (yet) supported.

3.4.7 interrupt

Scope

`Space::FooStreamDataReader`

Synopsis

```
#include <SpaceStreamsApi.h>

DDS::ReturnCode_t
interrupt();
```

Description

Interrupt a blocking get operation from a different thread.

Return Value

`ReturnCode_t` Possible return codes of the operation are: `DDS::RETCODE_OK`, `DDS::RETCODE_ERROR`.

Detailed Description

The get operation accepts a `timeout` parameter which causes the `FooStreamDataReader` to block until data becomes available. It can block indefinitely when an infinite timeout is supplied and data never becomes available because there are simply no compatible writers.

In such cases it can be desirable to interrupt the get operation from the application, i.e. for termination or reclaiming of resources.

The `interrupt` call triggers an internal `GuardCondition` by calling `DDS::GuardCondition::set_trigger_value(true)`. This causes the get operation to return with a `DDS::RETCODE_NO_DATA` result.

Return Code

The return code of this operation is determined by the result of `DDS::GuardCondition::set_trigger_value()`

DDS::RETCODE_OK The `GuardCondition` was triggered successfully

DDS::RETCODE_ERROR An internal error occurred

3.5 FooStreamFilterCallback Interface

Scope

`Space::FooStreamDataReader`

Synopsis

```
#include <SpaceStreamsApi.h>

boolean
a_filter(
   const Space::Foo &data)
```

Description

Function interface for filters that are passed to the get_w_filter and/or `peek_w_filter` operations.

Parameters

in const Foo &data A data sample.

Return Value

boolean Return `true` if the supplied data matches, `false` if it doesn't match.

Detailed Description

The application can supply any function that adheres to the `FooStreamFilterCallback` interface, to filter data that is retrieved by the get_w_filter operation. If the data matches the filter, the function returns `true` and the data is added to the `data_values` buffer that is returned by the get_w_filter operation. Data that doesn't match the filter is discarded.

4

Contacts & Notices

4.1 Contacts

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA
Tel: +1 781 569 5819

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK
Tel: +44 (0)191 497 9900

PrismTech France
28 rue Jean Rostand
91400 Orsay
France
Tel: +33 (1) 69 015354

Web: http://www.prismtech.com

E-mail: info@prismtech.com

4.2 Notices

This document may be reproduced in whole but not in part. The information contained in this document is subject to change without notice and is made available in good faith without liability on the part of PrismTech Limited or PrismTech Corporation. All trademarks acknowledged.