RTI Connext Micro

API and QoS Guide

Version 2.4.1

Your systems. Working as one.

Trademarks

Copy and Use Restrictions

Technical Support

Real-Time Innovations, Inc.
232 E. Jave Drive
Sunnyvale, CA 94089

Phone:	(408) 990-7444
Email:	support@rti.com
Website:	http://www.rti.com

Contents

RTI Connext Micro API and QoS Guide

This guide presents the APIs and Quality of Service (QoS) supported by *RTI Connext Micro* (formerly *RTI Micro Edition*). It assumes familiarity with *Connext DDS*, *Connext Messaging*, or the standard DDS API and QoS.

As a DDS-compliant implementation with a small footprint, implements a subset of the standard DDS API. A baseline API is provided by default that provides functionality for:

❏ Creating and deleting DDS entities
❏ Writing and receiving data
❏ Configuring notifications for statuses via listeners

When using the available buildable source code, some APIs may be optionally enabled or disabled at compile time. In addition to this guide, please consult the Release Notes (**RTI_Connext_Micro_ReleaseNotes.pdf**) and the API Reference HTML documentation for details on how to configure these optional APIs.

1 Baseline API and QoS Policies

The baseline configuration of *RTI Connext Micro* provides the minimal subset of APIs and QoS needed to create (and delete) an application that can publish and subscribe to other DDS-compatible applications.

This section describes the subset of the *Connext* API that is available in the baseline configuration of *RTI Connext Micro*.

For APIs, function declarations are provided below. Further descriptions are provided in HTML documentation. For complete function prototypes and descriptions for all DDS APIs, see the *RTI Connext Core Libraries and Utilities* documentation.

1.1 Supported C APIs

The following are the supported C language DDS APIs.

Note: some platforms restrict calling APIs that free dynamically allocated memory. Because finalize and delete operations in general can free memory, some of the APIs listed in this document may either be not available or are no-ops for such platforms. See HTML documentation for allowed APIs for specific platforms and also safety certifiable versions of *RTI Connext Micro*.

1.1.1 Creating Entities and Registering Types

DDS Entities are created one at a time, and either auto-enabled upon creation or explicitly enabled after creation. Topics are created for user data types that first must be registered.

- ❏ DDS_DomainParticipantFactory_create_participant()
- ❏ DDS_DomainParticipantFactory_get_instance()
- ❏ DDS_DomainParticipant_create_publisher()
- ❏ DDS_DomainParticipant_create_subscriber()
- ❏ DDS_DomainParticipant_register_type()
- ❏ DDS_DomainParticipant_create_topic()
- ❏ DDS_Publisher_create_datawriter()
- ❏ DDS_Subscriber_create_datareader()

1.1.2 Enabling Entities

Enabling an entity after its creation is done with

- ❏ DDS_Entity_enable

To use the above function, each entity must be cast to a DDS_Entity, and this is done with the following:

- ❏ DDS_DomainParticipant_as_entity()
- ❏ DDS_Publisher_as_entity()
- ❏ DDS_Subscriber_as_entity()
- ❏ DDS_DataWriter_as_entity()
- ❏ DDS_DataReader_as_entity()
- ❏ DDS_Topic_as_entity()

1.1.3 Configuring DomainParticipantFactory QoS

The QoS for each entity can be specified as input when the entity is created. However, since the DomainParticipantFactory is not an “entity,” its QoS policies are configured with these APIs:

- ❏ DDS_DomainParticipantFactory_get_qos()
- ❏ DDS_DomainParticipantFactory_set_qos()

1.1.4 Finalizing QoS

Upon shutdown, the QoS for the DomainParticipantFactory and each entity should be finalized with the following APIs. This ensures that internal resources are properly cleaned up.

- ❏ DDS_DataReaderQos_finalize()
- ❏ DDS_DataWriterQos_finalize()
- ❏ DDS_DomainParticipantFactoryQos_finalize()
- ❏ DDS_DomainParticipantQos_finalize()
- ❏ DDS_PublisherQos_finalize()
- ❏ DDS_SubscriberQos_finalize()
- ❏ DDS_TopicQos_finalize()

1.1.5 Deleting Entities

Entities may delete all of their contained entities at once by calling these APIs:

- ❏ DDS_DomainParticipant_delete_contained_entities()
- ❏ DDS_Publisher_delete_contained_entities()
- ❏ DDS_Subscriber_delete_contained_entities()

To clean up the DomainParticipantFactory’s internal resources, use this:

- ❏ DDS_DomainParticipantFactory_finalize_instance()

Alternatively, individual entities can be deleted with these APIs:

- ❏ DDS_DomainParticipant_delete_publisher()
- ❏ DDS_DomainParticipant_delete_subscriber()
- ❏ DDS_DomainParticipant_delete_topic()
- ❏ DDS_DomainParticipantFactory_delete_participant()
- ❏ DDS_Publisher_delete_datawriter()
- ❏ DDS_Subscriber_delete_datareader()

Finally, registered types can be unregistered:

- ❏ DDS_DomainParticipant_unregister_type()

1.1.6 Getting Entities

An entity can access its parent entity. DataWriters and DataReaders can also access their Topics and Topic descriptions, respectively.

- ❏ DDS_DataReader_get_subscriber()
- ❏ DDS_DataReader_get_topicdescription()
- ❏ DDS_DataWriter_get_publisher()
- ❏ DDS_DataWriter_get_topic()
- ❏ DDS_Publisher_get_participant()
- ❏ DDS_Subscriber_get_participant()
- ❏ DDS_TopicDescription_get_name()
- ❏ DDS_TopicDescription_get_participant()
- ❏ DDS_TopicDescription_get_type_name()

1.1.7 Getting Topics

A Topic or Topic description can be obtained from its DomainParticipant.

- ❏ DDS_DomainParticipant_find_topic() (*Note: timeout not supported)
- ❏ DDS_DomainParticipant_lookup_topicdescription()

1.1.8 Writing, Registering, and Disposing Data

Data can be written with or without a user-specified timestamp.

- ❏ DDS_DataWriter_write()
- ❏ DDS_DataWriter_write_w_timestamp()

Keyed data can have instances be registered, unregistered, and disposed.

- ❏ DDS_DataWriter_register_instance()
- ❏ DDS_DataWriter_register_instance_w_timestamp()
- ❏ DDS_DataWriter_unregister_instance()
- ❏ DDS_DataWriter_unregister_instance_w_timestamp()
- ❏ DDS_DataWriter_dispose()
- ❏ DDS_DataWriter_dispose_w_timestamp()

All above functions can be called by a typed DataWriter. For example, the functions FooDataWriter_write(), FooDataWriter_register_instance(), and FooDataWriter_dispose() will be among the functions available for user data type Foo.

1.1.9 Receiving Data

Received data can be accessed one sample at a time, either by reading or taking. If taking, the loan on the sample's resources should eventually be returned.

❑ DDS_DataReader_read()
❑ DDS_DataReader_read_instance()
❑ DDS_DataReader_read_next_sample()
❑ DDS_DataReader_return_loan()
❑ DDS_DataReader_take()
❑ DDS_DataReader_take_instance()
❑ DDS_DataReader_take_next_sample()

These read, take, and return loan functions can also be called by a typed DataReader; for example: FooDataReader_take_next_sample().

1.1.10 DataWriter and DataReader Listeners

Listeners are available for DataWriters and DataReaders, and a few statuses are handled by the baseline configuration. First, matching of local DataWriters with remote DataReaders, and vice versa, is necessary to ensure the initial discovery phase has successfully completed; thus, the on_publication_matched() and on_subscription_matched() callbacks are provided.

Next, since DataWriters and DataReaders may communicate with remote entities using non-default QoS, it is necessary to know whether QoS incompatibilities exist; thus, the on_offered_incompatible_qos() callback is provided. Lastly, a DataReader should notify its user when data is available to be read or taken; thus, the on_data_available() DataReaderListener callback is provided.

In summary, a DataWriterListener has these callbacks:

❑ on_offered_incompatible_qos()
❑ on_publication_matched()
❑ on_offered_deadline_missed()
❑ on_liveliness_lost()
❑ on_reliable_reader_activity_changed()

A DataReaderListener has these callbacks:

- ❏ on_data_available()
- ❏ on_requested_incompatible_qos()
- ❏ on_subscription_matched()
- ❏ on_sample_rejected()
- ❏ on_sample_lost()
- ❏ on_liveliness_changed()
- ❏ on_requested_deadline_missed()
- ❏ on_instance_replaced()

In addition, two other DataReaderListener callbacks enable an application to filter samples based on content:

- ❏ on_before_sample_commit()
- ❏ on_before_sample_deserialize()

Each is called when a sample is received by the DataReader. on_before_sample_deserialize() is called before the (serialized) sample has been deserialized. on_before_sample_commit() is called after the sample has been serialized but before it has been made available for read or take. Each callback allows the application to determine, from either serialized or deserialized data, whether the sample will be dropped, thus acting as a content-based filtering listener callback.

The HelloWorld and HelloWorld_dpde examples show how to use these callbacks for filtering samples. Consult the API Reference HTML documentation for further details.

1.2 Supported C++ Classes and Methods

Documentation of C++ APIs, an early access feature in this release, is provided in HTML documentation, in module *RTI Connext Micro C++ API Reference*.

1.3 Supported QoS Policies

The baseline configuration of *RTI Connext Micro* uses DDS-standard default QoS values. Any deviations from the DDS default values, as well as any unsupported QoS policies or fields, are described below.

QoS policies not described below should be assumed to be supported only for default values.

A general note on interoperability: because *RTI Connext Micro* does not support all fields for all QoS, it may not have the same default values as other DDS implementations. Thus, requested-offered (RxO) semantics need to be considered when interoperating with other DDS implementations. For example, *RTI Connext Micro* currently supports only MANUAL_BY_TOPIC LivelinessQos kind; to be compatible with a *RTI Connext Micro* DataReader, a DataWriter must also set its LivelinessQos kind to be MANUAL_BY_TOPIC, which is different than the DDS default kind of AUTOMATIC.

Inline QoS: QoS sent in-line with data is accepted, but is only interpreted if the QoS policy is supported.

1.3.1 DataReaderResourceLimits

The following DataReaderResourceLimits fields are supported. All must be finite values greater than 0.

- ❏ max_remote_writers
- ❏ max_remote_writers_per_instance
- ❏ max_samples_per_remote_writer
- ❏ max_outstanding_reads

1.3.2 DataWriterResourceLimits

DataWriterResourceLimits QoS policy is supported with field max_remote_readers.

1.3.3 Destination Order

Only the BY_RECEPTION_TIMESTAMP kind is supported.

1.3.4 Durability

Only VOLATILE and TRANSIENT_LOCAL Durability kinds are supported.

1.3.5 History

Only KEEP_LAST History kind is supported.

1.3.6 LatencyBudget

The LatencyBudget QoS policy is not supported (effectively zero duration).

1.3.7 Lifespan

The Lifespan QoS policy is not supported (effectively infinite duration).

1.3.8 Liveliness

When lease duration is infinite, any Liveliness kind may be set. When lease duration is finite, only the MANUAL_BY_TOPIC Liveliness kind is supported. Lease duration has range of [1, infinite].

1.3.9 Deadline

The Deadline Qos policy is supported.

1.3.10 Ownership and OwnershipStrength

Both SHARED and EXCLUSIVE Ownership kinds are supported.

1.3.11 Presentation

Coherent sets are not supported, therefore the coherent_access field is not supported.

1.3.12 Reliability

Both BEST-EFFORT and RELIABLE Reliability kinds are supported.

For RELIABLE reliability, additional settings related to the RTPS protocol (e.g. periodic and piggyback HEARTBEATs) are available, see HTML documentation for specifics.

1.3.13 ResourceLimits

The following limits in the ResourceLimits QoS policy are supported. All must be finite values greater than 0.

- max_samples
- max_instances
- max_samples_per_instance

Note that max_samples must be large enough to contain all samples for all instances:

```
max_samples >= max_instances * max_samples_per_instance
```

1.3.14 TimeBasedFilter

The TimeBasedFilter QoS policy is not supported.

1.4 ContentFilteredTopics

ContentFilteredTopics are not supported, but content filtering is provided by two DataReaderListener callbacks: on_before_sample_deserialize() and on_before_sample_commit() (see Section 1.1.10).

1.5 Conditions and WaitSets

StatusConditions, GuardConditions, and WaitSets are supported.

2 Disabling Optional APIs and QoS Policies

When building *RTI Connext Micro* from buildable source, some of the APIs and QoS available in *RTI Connext Micro* may optionally be removed. Predefined compiler flags may be selectively disabled for different features; this can minimize the library size by removing unused functionality.

2.1 C APIs

The following APIs are enabled in *RTI Connext Micro*. A custom build can disable the corresponding flags to remove these APIs.

2.1.1 Configuring QoS (INCLUDE_API_QOS)

The following APIs are enabled by default. If QoS configuration is not necessary, these APIs can be disabled in source code by defining the preprocessor flag INCLUDE_API_QOS as 0.

- DDS_DataReader_get_qos()
- DDS_DataReader_set_qos()
- DDS_DataWriter_get_qos()
- DDS_DataWriter_set_qos()
- DDS_DomainParticipant_get_default_publisher_qos()
- DDS_DomainParticipant_get_default_subscriber_qos()
- DDS_DomainParticipant_get_default_topic_qos()
- DDS_DomainParticipant_get_qos()
- DDS_DomainParticipant_set_default_publisher_qos()
- DDS_DomainParticipant_set_default_subscriber_qos()
- DDS_DomainParticipant_set_default_topic_qos()
- DDS_DomainParticipant_set_qos()
- DDS_DomainParticipantFactory_get_default_participant_qos()

- DDS_DomainParticipantFactory_set_default_participant_qos()
- DDS_Publisher_copy_from_topic_qos()
- DDS_Publisher_get_default_datawriter_qos()
- DDS_Publisher_get_qos()
- DDS_Publisher_set_default_datawriter_qos()
- DDS_Publisher_set_qos()
- DDS_Subscriber_copy_from_topic_qos()
- DDS_Subscriber_get_default_datareader_qos()
- DDS_Subscriber_get_qos()
- DDS_Subscriber_set_default_datareader_qos()
- DDS_Subscriber_set_qos()
- DDS_Topic_get_qos()
- DDS_Topic_set_qos()

2.1.2 Looking Up Local and Remote Entities (INCLUDE_API_LOOKUP)

The following APIs are used to take inventory of and gain access to created local entities. They are enabled by default. If they are not needed, they can be disabled in source code by defining the preprocessor flag INCLUDE_API_LOOKUP as 0.

- DDS_DomainParticipant_find_topic()
- DDS_DomainParticipant_lookup_topicdescription()
- DDS_Publisher_lookup_datawriter()
- DDS_Subscriber_lookup_datareader()
- DDS_DataReader_lookup_instance()

Note lookup_instance can be called by a typed DataReader, as FooDataReader_lookup_instance().

2.2 QoS

RTI Connext Micro does not have any QoS that may optionally be disabled at compile time by setting a predefined flag.

3 Transports

RTI Connext Micro supports a built-in UDPv4 transport for inter-process communication. In addition, an intra-process transport is supported for communication between entities within a single DomainParticipant, as an alternative to UDPv4 loopback.

No additional transports are currently supported. This includes the transports available from the *RTI Limited Bandwidth Plug-Ins*.

4 Discovery

RTI Connext Micro supports both dynamic and static endpoint discovery:

- ❏ **Static**. Remote endpoint state is manually configured and asserted by the user and only those asserted remote endpoints may be discovered.
- ❏ **Dynamic**. Remote endpoint state is automatically propagated between built-in discovery writers and readers.

Example HelloWorld source code is available for each discovery type: **HelloWorld** for static, and **HelloWorld_dpde** for dynamic.

Note: In both cases, participant discovery is dynamic, handled automatically between participants.

5 User-Data Type-Support Code Generation

User-defined data types need type-support code to interface with *RTI Connext Micro.* This type-support code can be generated automatically by using the *rtiddsgen* utility.

rtiddsgen takes an input type-definition in IDL and produces type-support code that *RTI Connext Micro* uses when publishing and subscribing to data of the type.

To run *rtiddsgen,* invoke the script in **<install_dir>/rtiddsgen/scripts**. Given an example type, **Foo.idl**, the following command will generate type-support code (while replacing any previously generated code):

```
rtiddsgen -micro -language C -replace Foo.idl
```

For C++, the language option must be C++:

```
rtiddsgen -micro -language C++ -replace Foo.idl
```

See the *RTI Connext Micro Release Notes* for the latest supported types and features of *rtiddsgen*.

6 Wire Interoperability

RTI Connext Micro uses a subset of the OMG Real-Time Publish-Subscribe (RTPS) wire protocol, version . It is interoperable with applications implemented using *RTI Connext Messaging* or *Connext DDS*, *RTI Data Distribution Service* 4.5, as well as applications built using DDS implementations from other vendors.

RTI Connext Micro supports serialization and deserialization of samples. It does not support RTPS fragmented data samples; therefore samples are limited to the transport's maximum message size.

7 Example Code

RTI Connext Micro includes source code for a few examples. While the baseline API of *Connext Micro* is shared with *Connext Messaging* and *Connext DDS*, there are APIs unique to *RTI Connext Micro.* A few APIs are outlined below.

7.1 Managing Writer and Reader History Plug-ins

Queues for storing written and received samples are managed by the Writer History and Reader History factories, respectively. *RTI Connext Micro* provides pluggable interfaces for both, so each needs to be registered (and unregistered) with the DomainParticipantFactory's registry.

```
/* Register Writer and Reader Histories */
RTComponentFactoryRegistry_t *registry;

registry =
        DDS_DomainParticipantFactory_get_registry
        (DDS_DomainParticipantFactory_get_instance());

RTComponentFactoryRegistry_register(
        registry, “wh”,
        RTI_SmWriterHistoryFactory_get_interface()
        NULL, NULL);

    RTComponentFactoryRegistry_register(
        registry, “rh”,
        RTI_SmReaderHistoryFactory_get_interface()
        NULL, NULL);
```

7.2 Configuring Transport Allowed Interfaces

A transport component module, such as UDP, has a set of allowed interfaces, over which data is allowed to be sent and received.

An example configuration of allowed interfaces for the UDP transport:

```
/* Re-register UDP component with updated allowed interfaces */
RTComponentFactoryRegistry_t *registry;
struct UDPInterfaceFactoryProperty *udp_property;
...

registry =
        DDS_DomainParticipantFactory_get_registry
        (DDS_DomainParticipantFactory_get_instance());

/* First, unregister UDP */
if (!RTComponentFactoryRegistry_unregister(
    registry, "_udp", NULL, NULL))
    {
        printf("failed to unregister udp\n");
        goto done;
    }

udp_property = (struct UDPInterfaceFactoryProperty *)
    malloc(sizeof(struct UDPInterfaceFactoryProperty));
*udp_property = UDPINTERFACE_FACTORY_PROPERTY_DEFAULT;

/* Set allowed_interface in property */
REDAStringSeq_set_maximum(&udp_property->allow_interface,2);
REDAStringSeq_set_length(&udp_property->allow_interface,2);

*REDAStringSeq_get_reference(&udp_property->allow_interface,0) =
    "lo";
*REDAStringSeq_get_reference(&udp_property->allow_interface,1) =
    "eth0";
```

```
/* Re-register UDP with updated property */
if (!RTComponentFactoryRegistry_register(registry, "_udp",
    UDPInterfaceFactory_get_interface(),
    (struct RTComponentFactoryProperty*)udp_property, NULL))
    {
       printf("failed to register udp\n");
       goto done;
   }
```

7.3 Managing Discovery Plug-ins

Discovery state is managed by the Discovery Factory. The factory can be configured for static or dynamic endpoint discovery. *RTI Connext Micro* provides a pluggable interface for discovery, so it needs to be registered with the DomainParticipantFactory. The example below registers the static endpoint discovery plugin:

```
/* Setting up Static Endpoint Discovery Plugin */
struct DDS_DomainParticipantQos participant_qos =
    DDS_DomainParticipantQos_INITIALIZER;
struct DPSE_Discovery_PluginProperty
    discovery_plugin_properties =
    DPSE_Discovery_PluginProperty_INITIALIZER;
RTComponentFactoryRegistry_t *registry;
...
registry =
        DDS_DomainParticipantFactory_get_registry
        (DDS_DomainParticipantFactory_get_instance());

RTComponentFactoryRegistry_register(
        registry, “dpse”,
        DPSE_Discovery_Factory_get_interface(),
        &discovery_plugin_properties._parent,
        NULL);

OSAPIStdio_snprintf(
    participant_qos.discovery.discovery.name,8,”dpse”);

/* Setting initial peers */
const char *peer = “10.10.1.123”;
DDS_StringSeq_set_maximum(
    &participant_qos.discovery.initial_peers,1);
DDS_StringSeq_set_length(
    &participant_qos.discovery.initial_peers,1);
*DDS_StringSeq_get_reference(
        &participant_qos.discovery.initial_peers,0) =
        DDS_String_dup(peer);
```

7.4 Configuring Static Discovery

With static endpoint discovery, remote endpoints must be asserted in order to be discoverable. Assuming a local DataWriter, the remote DataReader's subscription built-in topic data will need to be configured, then used to assert the remote subscription. Similar configuration must be done for remote publications of a local DataReader.

The remote endpoint configuration must be the same as the configuration of the local endpoint. For example, given a DataWriter, it must assert a remote DataReader with the same QoS, topic, type, and other state that was used to create the DataReader.

We show this with the example source code below.

A DataReader is created with its QoS configuration set to include RTPS object ID 200 and Reliability QoS kind set to RELIABLE:

```
/* Creating local DataReader */
DDS_DataReader *datareader = NULL;
struct DDS_DataReaderQos dr_qos = DDS_DataReaderQos_INITIALIZER;

...

/* Set Reader identity */
dr_qos.protocol.rtps_object_id = 200;
dr_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
...
datareader = DDS_Subscriber_create_datareader(subscriber,
                 DDS_Topic_as_topicdescription(application->topic),
                 &dr_qos,
                 &dr_listener,
                 DDS_DATA_AVAILABLE_STATUS);
```

For a DataWriter to match the above DataReader, it must configure the remote subscription built-in topic data to be the same:

```
/* DataWriter asserting remote DataReader */
struct DDS_SubscriptionBuiltinTopicData rem_subscription_data =
    DDS_SubscriptionBuiltinTopicData_INITIALIZER;

...

/* Configure remote reader */
rem_subscription_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] =
    200;
```

```
rem_subscription_data.reliability.kind =
    DDS_RELIABLE_RELIABILITY_QOS;
...
if (DDS_RETCODE_OK !=
    DPSE_RemoteSubscription_assert(participant,
                                   "subscriber",
                                   &rem_subscription_data,
                                   NDDS_TYPEPLUGIN_NO_KEY)) {
    printf("failed to assert remote subscription\n");
}
```

7.5 Type Support

7.5.1 Generating Type-Support Code with rtiddsgen

Support code for user-defined data types can be generated by the *rtiddsgen* utility.

For an example data type defined in the IDL file, **Foo.idl**, *rtiddsgen* will generate type-support code with the command:

```
rtiddsgen -micro -language C -replace Foo.idl
```

7.5.2 Registering a Type

To be used by *RTI Connext Micro*, a type must be registered with the DomainParticipant. The generic type registration function is as follows:

```
retcode = DDS_DomainParticipant_register_type(
   participant,type_name,FooTypeSupport_get_instance());
```

8 Logging API

RTI Connext Micro maintains a log of events occurring in an application and provides APIs to configure the log buffer and process log messages.

For descriptions of these APIs, users should consult online documentation in the module *RTI Connext Micro Logging Reference*. It describes:

- ❑ The format and kinds of log messages
- ❑ Managing the size of the log buffer
- ❑ Interpreting Log Message Kinds and Error Codes

- Setting a Log Handler Callback Function for user-specific processing of log messages