RTI Connext DDS Micro

User's Manual

Version 2.4.12

Contents:

RTI® Connext® DDS Micro provides a small-footprint, modular messaging solution for resource-limited devices that have limited memory and CPU power, and may not even be running an operating system. It provides the communications services that developers need to distribute time-critical data. Additionally, *Connext DDS Micro* is designed as a certifiable component in high-assurance systems.

Key benefits of *Connext DDS Micro* include:

- Accommodations for resource-constrained environments.
- Modular and user extensible architecture.
- Designed to be a certifiable component for safety-critical systems.
- Seamless interoperability with *RTI Connext DDS Professional*.

Chapter 1

Introduction

1.1 What is RTI Connext DDS Micro?

RTI Connext DDS Micro is network middleware for distributed real-time applications. It provides the communications service programmers need to distribute time-critical data between embedded and/or enterprise devices or nodes. *Connext DDS Micro* uses the publish-subscribe communications model to make data distribution efficient and robust. *Connext DDS Micro* simplifies application development, deployment and maintenance and provides fast, predictable distribution of time-critical data over a variety of transport networks. With *Connext DDS Micro*, you can:

- Perform complex one-to-many and many-to-many network communications.
- Customize application operation to meet various real-time, reliability, and quality-of-service goals.
- Provide application-transparent fault tolerance and application robustness.
- Use a variety of transports.

Connext DDS Micro implements the Data-Centric Publish-Subscribe (DCPS) API within the OMG's Data Distribution Service (DDS) for Real-Time Systems. DDS is the first standard developed for the needs of real-time systems. DCPS provides an efficient way to transfer data in a distributed system.

With *Connext DDS Micro*, systems designers and programmers start with a fault-tolerant and flexible communications infrastructure that will work over a wide variety of computer hardware, operating systems, languages, and networking transport protocols. *Connext DDS Micro* is highly configurable so programmers can adapt it to meet the application's specific communication requirements.

1.1.1 RTI Connext DDS Cert versus RTI Connext DDS Micro

RTI Connext DDS Micro and *RTI Connext DDS Cert* originate from the same source base, but as of *Connext DDS Micro* 2.4.6 the two are maintained as two independent releases. The latest release with certification evidence is *Connext DDS Cert* 2.4.5. However, features that exist in *Connext DDS Micro* and *Connext DDS Cert* behave identically and the source code is written following identical guidelines. *Connext DDS Cert* only supports a subset of the features found in *Connext*

DDS Micro. In the API reference manuals, APIs that are supported by *Connext DDS Cert* are clearly marked.

1.1.2 Optional Certification Package

An optional Certification Package is available for systems that require certification to DO-178C or other safety standards. This package includes the artifacts required by a certification authority. The Certification Package is licensed separately from Connext DDS Cert.

To use an existing Certification Package, an application must be linked against the same libraries included in the Certification Package. Contact RTI Support, support@rti.com, for details.

1.1.3 Publish-Subscribe Middleware

Connext DDS Micro is based on a publish-subscribe communications model. Publish-subscribe (PS) middleware provides a simple and intuitive way to distribute data. It decouples the software that creates and sends data—the data publishers—from the software that receives and uses the data—the data subscribers. Publishers simply declare their intent to send and then publish the data. Subscribers declare their intent to receive, then the data is automatically delivered by the middleware. Despite the simplicity of the model, PS middleware can handle complex patterns of information flow. The use of PS middleware results in simpler, more modular distributed applications. Perhaps most importantly, PS middleware can automatically handle all network chores, including connections, failures, and network changes, eliminating the need for user applications to program of all those special cases. What experienced network middleware developers know is that handling special cases accounts for over 80% of the effort and code.

1.2 Supported DDS Features

Connext DDS Micro supports a subset of the DDS DCPS standard. A brief overview of the supported features are listed here. For a detailed list, please refer to the C API Reference and C++ API Reference.

1.2.1 DDS Entity Support

Connext DDS Micro supports the following DDS entities. Please refer to the documentation for details.

- DomainParticipantFactory
- DomainParticipant
- Topic
- Publisher
- Subscriber
- DataWriter
- DataReader

1.2.2 DDS QoS Policy Support

Connext DDS Micro supports the following DDS Qos Policies. Please refer to the documentation for details.

- DDS_DataReaderProtocolQosPolicy
- DDS_DataReaderResourceLimitsQosPolicy
- DDS_DataWriterProtocolQosPolicy
- DDS_DataWriterResourceLimitsQosPolicy
- DDS_DeadlineQosPolicy
- DDS_DiscoveryQosPolicy
- DDS_DomainParticipantResourceLimitsQosPolicy
- DDS_DurabilityQosPolicy
- DDS_DestinationOrderQosPolicy
- DDS_EntityFactoryQosPolicy
- DDS_HistoryQosPolicy
- DDS_LivelinessQosPolicy
- DDS_OwnershipQosPolicy
- DDS_OwnershipStrengthQosPolicy
- DDS_ReliabilityQosPolicy
- DDS_ResourceLimitsQosPolicy
- DDS_RtpsReliableWriterProtocol_t
- DDS_SystemResourceLimitsQosPolicy
- DDS_TransportQosPolicy
- DDS_UserTrafficQosPolicy
- DDS_WireProtocolQosPolicy

1.3 Standards and Interoperability

Connext DDS Micro implements the Object Management Group (OMG) Data Distribution Service (DDS) standard (version 1.4), and the Real-Time Publish-Subscribe (RTPS) wire interoperabilty protocol standard (version 2.2).

Connext DDS Micro supports a subset of the submessages defined by the Real-Time Publish-Subscribe (RTPS) interoperability specification. Data fragment submessages are not supported. The messages are compatible with Wireshark and its RTPS packet dissector.

Connext DDS Micro and *Connext DDS* are wire-interoperable, unless stated otherwise (see below), and API compatible for APIs specified by the DDS standard. For non-standard APIs, *Connext*

DDS Micro and *Connext DDS* are incompatible. Please refer to *Working with RTI Connext DDS Micro and RTI Connext DDS* for more information.

1.3.1 DDS Wire Compatability

Connext DDS Micro is compliant with RTPS 2.2, but does not support and ignore the following RTPS sub-messages:

Submessage	Supported	DDS Standard	Connext DDS Core
DATA_FRAG	No	Yes	Yes
NACK_FRAG	No	Yes	Yes
HEARTBEAT_FRAG	No	Yes	No
INFO_SRC	No	Yes	Yes
INFO_REPLY	No	Yes	Yes
INFO_REPLY_IPV4	No	Yes	Yes

1.3.2 Profile / Feature

Connext DDS Micro does not support mutable Qos policies.

Submessage	Supported	DDS Standard	Connext DDS Core
USER_DATA	No	Yes	Yes
TOPIC_DATA	No	Yes	Yes
DURABILITY	Partially (1)	Yes	Yes
PRESENTATION	Partially (2)	Yes	Yes
DEADLINE	Yes	Yes	Yes
LATENCY_BUDGET	No	Yes	Yes
LIVELINESS	Partially (3)	Yes	Yes
TIME_BASED_FILTER	No	Yes	Yes
PARTITION	No	Yes	Yes
RELIABILITY	Yes (4)	Yes	Yes
TRANSPORT_PRIORITY	No	Yes	Yes
LIFESPAN	No	Yes	Yes
DESTINATION_ORDER	Partially (5)	Yes	Yes
HISTORY	Partially (6)	Yes	Yes
RESOURCE_LIMITS	Yes (7)	Yes	Yes
ENTITY_FACTORY	Yes	Yes	Yes
WRITER_DATA_LIFECYCLE	No	Yes	Yes
READER_DATA_LIFECYCLE	No	Yes	Yes
OWNERSHIP	Yes	Yes	Yes
OWNERSHIP_STRENGTH	Yes	Yes	Yes
DURABILITY_SERVICE	No	Yes	Yes
ContentFilteredTopic	No	Yes	Yes
QueryCondition	No	Yes	Yes
MultiTopic	No	Yes	No
ASYNCHRONOUS_PUBLISHER	No	No	Yes

Continued on next page

Table 1.1 – continued from previous page

Submessage	Supported	DDS Standard	Connext DDS Core
AVAILABILITY	No	No	Yes
BATCH	Only reception	No	Yes
DATA_READER_PROTOCOL	rtps_object_id	No	Yes
DATA_WRITER_PROTOCOL	Partially (8)	No	Yes
DISCOVERY	Yes	No	Yes
DISCOVERY_CONFIG	No	No	Yes
ENTITY_NAME	Partially (9)	No	Yes
EVENT	No	No	Yes
LOCATORFILTER	No	No	Yes
LOGGING	No	No	Yes
MULTICHANNEL	No	No	Yes
PROPERTY	No	No	Yes
PUBLISH_MODE	No	No	Yes
RECEIVER_POOL	No	No	Yes
SERVICE	No	No	Yes
TYPE_CONSISTENCY_ENFORCEMENT	No	No	Yes
TYPESUPPORT	Yes	No	Yes
WIRE_PROTOCOL	Yes	No	Yes

NOTES:

1. VOLATILE and TRANSIENT_LOCAL
2. No, DW offers access_scope = TOPIC, coherent_access = FALSE and ordered_access = TRUE DR requests access_scope = INSTANCE, coherent_access = FALSE and ordered_access = FALSE
3. AUTOMATIC (infinite only), MANUAL_BY_PARTICIPANT (infinite only), MANUAL_BY_TOPIC (finite and infinite)
4. BEST_EFFORT and RELIABLE, only max_blocking_time=0
5. DataWriter: Yes, DataReader only supports BY_RECEPTION_TIMESTAMP
6. Only KEEP_LAST
7. Only finite resource-limits
8. The following are supported: - heartbeat_period - heartbeats_per_max_samples - max_heartbeat_retries - max_send_window_size - rtps_object_id
9. DomainParticipant only

1.3.3 DDS API Support

For supported APIs, please refer to:

- C API Reference
- C++ API Reference

1.4 RTI Connext DDS Documentation

Throughout this document, we may suggest reading sections in other *RTI Connext DDS* documents. These documents are in your *RTI Connext DDS* installation directory under **rti-connext-dds-<version>/doc/manuals**. A quick way to find them is from *RTI Launcher's* Help panel, select "Browse Connext Documentation".

Since installation directories vary per user, links are not provided to these documents on your local machine. However, we do provide links to documents on the RTI Documentation site for users with Internet access.

New users can start by reading Parts 1 (Introduction) and 2 (Core Concepts) in the *RTI Connext DDS Core Libraries User's Manual*. These sections teach basic DDS concepts applicable to all RTI middleware, including *RTI Connext DDS Professional* and *RTI Connext DDS Micro*. You can open the *RTI Connext DDS Core Libraries User's Manual* from *RTI Launcher's* Help panel.

The RTI Community provides many resources for users of DDS and the RTI Connext family of products.

1.5 OMG DDS Specification

For the original DDS reference, the OMG DDS specification can be found in the OMG Specifications under "Data Distribution Service".

1.6 Other Products

RTI Connext DDS Micro is one of several products in the *RTI Connext* family of products:

RTI Connext DDS Cert is a subset of *RTI Connext DDS Micro*. *Connext DDS Cert* does not include the following features because Certification Evidence is not yet available for them. If you require Certification Evidence for any of these features, please contact RTI.

- C++ language API.
- Multi-platform support.
- Dynamic endpoint discovery.
- delete() APIs (e.g. delete_datareader())

RTI Connext DDS Professional addresses the sophisticated databus requirements in complex systems including an API compliant with the Object Management Group (OMG) Data Distribution Service (DDS) specification. DDS is the leading data-centric publish/subscribe (DCPS) messaging standard for integrating distributed real-time applications. *Connext DDS Professional* is the dominant industry implementation with benefits including:

- OMG-compliant DDS API
- Advanced features to address complex systems
- Advanced Quality of Service (QoS) support
- Comprehensive platform and network transport support

- Seamless interoperability with rtime

RTI Connext DDS Professional includes rich integration capabilities:

- Data transformation
- Integration support for standards including JMS, SQL databases, file, socket, Excel, OPC, STANAG, LabVIEW, Web Services and more
- Ability for users to create custom integration adapters
- Optional integration with Oracle, MySQL and other relational databases
- Tools for visualizing, debugging and managing all systems in real-time

RTI Connext DDS Professional also includes a rich set of tools to accelerate debugging and testing while easing management of deployed systems. These components include:

- Administration Console
- Distributed Logger
- Monitor
- Monitoring Library
- Recording Service

Chapter 2

Installation

2.1 Installing the RTI Connext DDS Micro Package

RTI Connext DDS Micro is provided in one of 2 zip files:

- rti_connext_dds_micro-2.4.12-Unix.zip
- rti_connext_dds_micro-2.4.12-Windows.zip

RTI Connext DDS Micro requires a Java Run-Time Environment (JRE) to run *rtiddsgen* and version 1.8.121 or better is required. Note that JRE 1.9 and higher is not supported. If a compatible JRE run-time envrionment is not already installed a compatible JRE can be installed from one of the following bundles:

- rti_connext_dds_micro-2.4.12-jre-darwin.zip – JRE for Darwin 32 and 64 Bit
- rti_connext_dds_micro-2.4.12-jre-i86Linux.zip – JRE for 32 bit Linux
- rti_connext_dds_micro-2.4.12-jre-i86Win32.zip – JRE for 32 bit Windows
- rti_connext_dds_micro-2.4.12-jre-x64Linux.zip – JRE for 64 bit Linux
- rti_connext_dds_micro-2.4.12-jre-x64Win64.zip – JRE for 64 bit Windows

Once installed, you will see a directory `rti_connext_dds_micro-2.4.12` in the installation directory. This installation directory contains this documentation, the *rtiddsgen* code generation tool, examples, and source code.

2.2 Setting Up Your Environment

The `RTIMEHOME` environment variable must be set to the installation directory path for *RTI Connext DDS Micro*.

2.3 Building Connext DDS Micro

This section is for users who are already familiar with CMake and may have built earlier versions of *Connext DDS Micro*. The sections following describe the process in detail and are recommended for everyone building *Connext DDS Micro*.

This section assumes that the *Connext DDS Micro* source-bundle has been downloaded and installed and that CMake is available.

1. Make sure CMake is in the path.
2. Run rtime-make.

   On UNIX® systems:

   ```
   cd <rti_me install directory>
   # you should see directories like doc/ lib/ rtiddsgen/ src/
   # and CMakeLists.txt

   resource/scripts/rtime-make --target self --name i86Linux4gcc7.3.0 \
               -G "Unix Makefiles" --build
   ```

   On Windows® systems:

   ```
   cd <rti_me install directory>
   # you should see directories like doc/ lib/ rtiddsgen/ src/
   # and CMakeLists.txt

   resource\scripts\rtime-make --target self --name i86Win32VS2015 \
               -G "NMake Makefiles" --build
   ```

3. You will find the *Connext DDS Micro* libraries here:

   On UNIX-based systems:

   ```
   # <rti_me install directory>/lib/i86Linux4gcc7.3.0
   ```

   On Windows systems:

   ```
   # <rti_me install directory>/lib/i86Win32VS2015
   ```

NOTE: `rtime-make` uses the platform specified with `--name` to determine a few settings needed by *Connext DDS Micro*. Please refer to *Preparing for a Build* for details.

For help, enter:

```
resource\scripts\rtime-make --help
```

To list available targets, enter:

```
resource\scripts\rtime-make --list
```

For help for a specific target, except self, enter:

```
resource\scripts\rtime-make --target <target> --help
```

Chapter 3

Getting Started

3.1 Define a Data Type

To distribute data using *Connext DDS Micro*, you must first define a data type, then run the *rtiddsgen* utility. This utility will generate the type-specific support code that *Connext DDS Micro* needs and the code that makes calls to publish and subscribe to that data type.

Connext DDS Micro accepts types definitions in Interface Definition Language (IDL) format.

For instance, the HelloWorld examples provided with *Connext DDS Micro* use this simple type, which contains a string "msg" with a maximum length of 128 chars:

```
struct HelloWorld {
       string<128> msg;
};
```

For more details, see *Data Types* in the *User's Manual*.

3.2 Generate Type Support Code with rtiddsgen

You will provide your IDL as an input to *rtiddsgen*. *rtiddsgen* supports code generation for the following standard types:

- octet, char, wchar
- short, unsigned short
- long, unsigned long
- long long, unsigned long long float
- double, long double
- boolean
- string
- struct
- array

- enum
- wstring
- sequence
- union
- typedef
- value type

The script to run *rtiddsgen* is in *<your_top_level_dir>/rti_connext_dds-6.0.0/rti_connext_micro-3.0.0/rtiddsgen/scripts*.

To generate support code for data types in a file called HelloWorld.idl:

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

Run `rtiddsgen -help` to see all available options. For the options used here:

- The `-micro` option is necessary to generate support code specific to *Connext DDS Micro*, as *rtiddsgen* is also capable of generating support code for *Connext DDS*, and the generated code for the two are different.
- The `-language` option specifies the language of the generated code. *Connext DDS Micro* supports C and C++ (with `-language C++`).
- The `-replace` option specifies that the new generated code will replace, or overwrite, any existing files with the same name.

rtiddsgen generates the following files for an input file HelloWorld.idl:

- **HelloWorld.h and HelloWorld.c**. Operations to manage a sample of the type, and a DDS sequence of the type.
- **HelloWorldPlugin.h and HelloWorldPlugin.c**. Implements the type-plugin interface defined by *Connext DDS Micro*. Includes operations to serialize and deserialize a sample of the type and its DDS instance keys.
- **HelloWorldSupport.h and HelloWorldSupport.c**. Support operations to generate a type-specific a *DataWriter* and *DataReader*, and to register the type with a DDS *DomainParticipant*.

This release of *Connext DDS Micro* supports a new way to generate support code for IDL Types that will generate a TypeCode object containing information used by an interpreter to serialize and deserialize samples. Prior to this release, the code for serialization and deserialization was generated for each type. To disable generating code to be used by the interpreter, use the `-interpreted 0` command-line option to generate code. This option generates code in the same way as was done in previous releases.

For more details, see *Generating Type Support with rtiddsgen* in the *User's Manual*.

3.3 Configure UDP Transport

You may need to configure the UDP transport component that is pre-registered by *RTI Connext DDS Micro*. To change the properties of the UDP transport, first the UDP component has be unregistered, then the properties have to be updated, and finally the component must be re-registered with the updated properties.

Example code:

- Unregister the pre-registered UDP component:

```
/* Unregister the pre-registered UDP component */
if (!RT_Registry_unregister(registry, "_udp", NULL, NULL))
{
    /* failure */
}
```

- Configure UDP transport properties:

```
struct UDP_InterfaceFactoryProperty *udp_property = NULL;

udp_property = (struct UDP_InterfaceFactoryProperty *)
    malloc(sizeof(struct UDP_InterfaceFactoryProperty));
if (udp_property != NULL)
{
    *udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    /* allow_interface: Names of network interfaces allowed to send/receive.
     * Allow one loopback (lo) and one NIC (eth0).
     */
    REDA_StringSeq_set_maximum(&udp_property->allow_interface,2);
    REDA_StringSeq_set_length(&udp_property->allow_interface,2);

    *REDA_StringSeq_get_reference(&udp_property->allow_interface,0) = DDS_String_
↪dup("lo");
    *REDA_StringSeq_get_reference(&udp_property->allow_interface,1) = DDS_String_
↪dup("eth0");
}
else
{
    /* failure */
}
```

- Re-register UDP component with updated properties:

```
if (!RT_Registry_register(registry, "_udp",
                     UDP_InterfaceFactory_get_interface(),
                    (struct RT_ComponentFactoryProperty*)udp_property, NULL))
{
    /* failure */
}
```

For more details, see the *Transports* section in the User's Manual.

3.4 Create DomainParticipant, Topic, and Type

A DomainParticipantFactory creates *DomainParticipants*, and a *DomainParticipant* itself is the factory for creating *Publishers*, *Subscribers*, and *Topics*.

When creating a *DomainParticipant*, you may need to customize DomainParticipantQos, notably for:

- **Resource limits**. Default resource limits are set at minimum values.
- **Initial peers**.
- **Discovery**. The name of the registered discovery component ("dpde" or "dpse") must be assigned to DiscoveryQosPolicy's name.
- **Participant Name**. Every *DomainParticipant* is given the same default name. Must be unique when using DPSE discovery.

Example code:

- Create a *DomainParticipant* with configured DomainParticipantQos:

```
DDS_DomainParticipant *participant = NULL;
struct DDS_DomainParticipantQos dp_qos =
     DDS_DomainParticipantQos_INITIALIZER;

/* DDS domain of DomainParticipant */
DDS_Long domain_id = 0;

/* Name of your registered Discovery component */
if (!RT_ComponentFactoryId_set_name(&dp_qos.discovery.discovery.name, "dpde"))
{
     /* failure */
}

/* Initial peers: use only default multicast peer */
DDS_StringSeq_set_maximum(&dp_qos.discovery.initial_peers,1);
DDS_StringSeq_set_length(&dp_qos.discovery.initial_peers,1);
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers,0) =
    DDS_String_dup("239.255.0.1");

/* Resource limits */
dp_qos.resource_limits.max_destination_ports = 32;
dp_qos.resource_limits.max_receive_ports = 32;
dp_qos.resource_limits.local_topic_allocation = 1;
dp_qos.resource_limits.local_type_allocation = 1;
dp_qos.resource_limits.local_reader_allocation = 1;
dp_qos.resource_limits.local_writer_allocation = 1;
dp_qos.resource_limits.remote_participant_allocation = 8;
dp_qos.resource_limits.remote_reader_allocation = 8;
dp_qos.resource_limits.remote_writer_allocation = 8;

/* Participant name */
strcpy(dp_qos.participant_name.name, "Participant_1");
```

(continues on next page)

(continued from previous page)

```
participant =
     DDS_DomainParticipantFactory_create_participant(factory,
                                                     domain_id,
                                                     &dp_qos,
                                                     NULL,
                                                     DDS_STATUS_MASK_NONE);
 if (participant == NULL)
 {
     /* failure */
 }
```

3.4.1 Register Type

Your data types that have been generated from IDL need to be registered with the *DomainParticipants* that will be using them. Each registered type must have a unique name, preferably the same as its IDL defined name.

```
DDS_ReturnCode_t retcode;

retcode = DDS_DomainParticipant_register_type(participant,
                                              "HelloWorld",
                                              HelloWorldTypePlugin_get());
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}
```

3.4.2 Create Topic of Registered Type

DDS *Topics* encapsulate the types being communicated, and you can create *Topics* for your type once your type is registered.

A topic is given a name at creation (e.g. "Example HelloWorld"). The type associated with the *Topic* is specified with its registered name.

```
DDS_Topic *topic = NULL;

topic = DDS_DomainParticipant_create_topic(participant,
                                           "Example HelloWorld",
                                           "HelloWorld",
                                           &DDS_TOPIC_QOS_DEFAULT,
                                           NULL,
                                           DDS_STATUS_MASK_NONE);

if (topic == NULL)
{
    /* failure */
}
```

3.4.3 DPSE Discovery: Assert Remote Participant

DPSE Discovery relies on the application to specify the other, or remote, *DomainParticipants* that its local *DomainParticipants* are allowed to discover. Your application must call a DPSE API for each remote participant to be discovered. The API takes as input the name of the remote participant.

```
/* Enable discovery of remote participant with name Participant_2 */
retcode = DPSE_RemoteParticipant_assert(participant, "Participant_2");
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}
```

For more information, see the *DDS Domains* section in the User's Manual.

3.5 Create Publisher

A publishing application needs to create a DDS *Publisher* and then a *DataWriter* for each *Topic* it wants to publish.

In *Connext DDS Micro*, PublisherQos in general contains no policies that need to be customized, while DataWriterQos does contain several customizable policies.

- Create *Publisher*:

```
DDS_Publisher *publisher = NULL;
publisher = DDS_DomainParticipant_create_publisher(participant,
                                                   &DDS_PUBLISHER_QOS_DEFAULT,
                                                   NULL,
                                                   DDS_STATUS_MASK_NONE);
if (publisher == NULL)
{
    /* failure */
}
```

For more information, see the *Sending Data* section in the User's Manual.

3.6 Create DataWriter

```
DDS_DataWriter *datawriter = NULL;
struct DDS_DataWriterQos dw_qos = DDS_DataWriterQos_INITIALIZER;
struct DDS_DataWriterListener dw_listener = DDS_DataWriterListener_INITIALIZER;

/* Configure writer Qos */
dw_qos.protocol.rtps_object_id = 100;
dw_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
dw_qos.resource_limits.max_samples_per_instance = 2;
dw_qos.resource_limits.max_instances = 2;
dw_qos.resource_limits.max_samples =
    dw_qos.resource_limits.max_samples_per_instance * dw_qos.resource_limits.max_
↪instances;
```

(continues on next page)

(continued from previous page)

```
dw_qos.history.depth = 1;
dw_qos.durability.kind = DDS_VOLATILE_DURABILITY_QOS;
dw_qos.protocol.rtps_reliable_writer.heartbeat_period.sec = 0;
dw_qos.protocol.rtps_reliable_writer.heartbeat_period.nanosec = 250000000;

/* Set enabled listener callbacks */
dw_listener.on_publication_matched = HelloWorldPublisher_on_publication_matched;

datawriter =
    DDS_Publisher_create_datawriter(publisher,
                                    topic,
                                    &dw_qos,
                                    &dw_listener,
                                    DDS_PUBLICATION_MATCHED_STATUS);
if (datawriter == NULL)
{
    /* failure */
}
```

The DataWriterListener has its callbacks selectively enabled by the DDS status mask. In the example, the mask has set the on_publication_matched status, and accordingly the DataWriterListener has its on_publication_matched assigned to a callback function.

```
void HelloWorldPublisher_on_publication_matched(void *listener_data,
                                                DDS_DataWriter * writer,
                                                const struct DDS_
↪PublicationMatchedStatus *status)
{
    /* Print on match/unmatch */
    if (status->current_count_change > 0)
    {
        printf("Matched a subscriber\n");
    }
    else
    {
        printf("Unmatched a subscriber\n");
    }
}
```

3.6.1 DPSE Discovery: Assert Remote Subscription

A publishing application using DPSE discovery must specify the other *DataReaders* that its *DataWriters* are allowed to discover. Like the API for asserting a remote participant, the DPSE API for asserting a remote subscription must be called for each remote *DataReader* that a *DataWriter* may discover.

Whereas asserting a remote participant requires only the remote *Participant*'s name, asserting a remote subscription requires more configuration, as all QoS policies of the subscription necessary to determine matching must be known and thus specified.

```
struct DDS_SubscriptionBuiltinTopicData rem_subscription_data =
    DDS_SubscriptionBuiltinTopicData_INITIALIZER;

/* Set Reader's protocol.rtps_object_id */
rem_subscription_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 200;

rem_subscription_data.topic_name = DDS_String_dup("Example HelloWorld");
rem_subscription_data.type_name = DDS_String_dup("HelloWorld");

rem_subscription_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

retcode = DPSE_RemoteSubscription_assert(participant,
                                         "Participant_2",
                                         &rem_subscription_data,
                                         HelloWorld_get_key_kind(HelloWorldTypePlugin_
↪get(),
                                         NULL)));
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}
```

3.6.2 Writing Samples

Within the generated type support code are declarations of the type-specific *DataWriter*. For the HelloWorld type, this is the HelloWorldDataWriter.

Writing a HelloWorld sample is done by calling the write API of the HelloWorldDataWriter.

```
HelloWorldDataWriter *hw_datawriter;
DDS_ReturnCode_t retcode;
HelloWorld *sample = NULL;

/* Create and set sample */
sample = HelloWorld_create();
if (sample == NULL)
{
    /* failure */
}
sprintf(sample->msg, "Hello World!");

/* Write sample  */
hw_datawriter = HelloWorldDataWriter_narrow(datawriter);

retcode = HelloWorldDataWriter_write(hw_datawriter, sample, &DDS_HANDLE_NIL);
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}
```

For more information, see the *Sending Data* section in the User's Manual.

3.7 Create Subscriber

A subscribing application needs to create a DDS *Subscriber* and then a *DataReader* for each *Topic* to which it wants to subscribe.

In *Connext DDS Micro*, SubscriberQos in general contains no policies that need to be customized, while DataReaderQos does contain several customizable policies.

```
DDS_Subscriber *subscriber = NULL;
subscriber = DDS_DomainParticipant_create_subscriber(participant,
                                                     &DDS_SUBSCRIBER_QOS_DEFAULT,
                                                     NULL,
                                                     DDS_STATUS_MASK_NONE);
if (subscriber == NULL)
{
    /* failure */
}
```

For more information, see the *Receiving Data* section in the User's Manual.

3.8 Create DataReader

```
DDS_DataReader *datareader = NULL;
struct DDS_DataReaderQos dr_qos = DDS_DataReaderQos_INITIALIZER;
struct DDS_DataReaderListener dr_listener = DDS_DataReaderListener_INITIALIZER;

/* Configure Reader Qos */
dr_qos.protocol.rtps_object_id = 200;
dr_qos.resource_limits.max_instances = 2;
dr_qos.resource_limits.max_samples_per_instance = 2;
dr_qos.resource_limits.max_samples =
    dr_qos.resource_limits.max_samples_per_instance * dr_qos.resource_limits.max_
↪instances;
dr_qos.reader_resource_limits.max_remote_writers = 10;
dr_qos.reader_resource_limits.max_remote_writers_per_instance = 10;
dr_qos.history.depth = 1;
dr_qos.durability.kind = DDS_VOLATILE_DURABILITY_QOS;
dr_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

/* Set listener callbacks */
dr_listener.on_data_available = HelloWorldSubscriber_on_data_available;
dr_listener.on_subscription_matched = HelloWorldSubscriber_on_subscription_matched;

datareader = DDS_Subscriber_create_datareader(subscriber,
                                              DDS_Topic_as_topicdescription(topic),
                                              &dr_qos,
                                              &dr_listener,
                                              DDS_DATA_AVAILABLE_STATUS | DDS_
↪SUBSCRIPTION_MATCHED_STATUS);
if (datareader == NULL)
{
```

(continues on next page)

(continued from previous page)

```
    /* failure */
}
```

The DataReaderListener has its callbacks selectively enabled by the DDS status mask. In the example, the mask has set the DDS_SUBSCRIPTION_MATCHED_STATUS and DDS_DATA_AVAILABLE_STATUS statuses, and accordingly the DataReaderListener has its on_subscription_matched and on_data_available assigned to callback functions.

```
void HelloWorldSubscriber_on_subscription_matched(void *listener_data,
                                                  DDS_DataReader * reader,
                                                  const struct DDS_
↪SubscriptionMatchedStatus *status)
{
    if (status->current_count_change > 0)
    {
        printf("Matched a publisher\n");
    }
    else
    {
        printf("Unmatched a publisher\n");
    }
}
```

```
void HelloWorldSubscriber_on_data_available(void* listener_data,
                                            DDS_DataReader* reader)
{
    HelloWorldDataReader *hw_reader = HelloWorldDataReader_narrow(reader);
    DDS_ReturnCode_t retcode;
    struct DDS_SampleInfo *sample_info = NULL;
    HelloWorld *sample = NULL;

    struct DDS_SampleInfoSeq info_seq =
        DDS_SEQUENCE_INITIALIZER(struct DDS_SampleInfo);
    struct HelloWorldSeq sample_seq =
        DDS_SEQUENCE_INITIALIZER(HelloWorld);

    const DDS_Long TAKE_MAX_SAMPLES = 32;
    DDS_Long i;

    retcode = HelloWorldDataReader_take(hw_reader,
       &sample_seq, &info_seq, TAKE_MAX_SAMPLES,
       DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);

    if (retcode != DDS_RETCODE_OK)
    {
        printf("failed to take data: %d\n", retcode);
        goto done;
    }

    /* Print each valid sample taken */
    for (i = 0; i < HelloWorldSeq_get_length(&sample_seq); ++i)
```

(continues on next page)

(continued from previous page)

```
    {
        sample_info = DDS_SampleInfoSeq_get_reference(&info_seq, i);

        if (sample_info->valid_data)
        {
            sample = HelloWorldSeq_get_reference(&sample_seq, i);
            printf("\nSample received\n\tmsg: %s\n", sample->msg);
        }
        else
        {
            printf("not valid data\n");
        }
    }

    HelloWorldDataReader_return_loan(hw_reader, &sample_seq, &info_seq);

 done:
    HelloWorldSeq_finalize(&sample_seq);
    DDS_SampleInfoSeq_finalize(&info_seq);
}
```

3.8.1 DPSE Discovery: Assert Remote Publication

A subscribing application using DPSE discovery must specify the other *DataWriters* that its *DataReaders* are allowed to discover. Like the API for asserting a remote participant, the DPSE API for asserting a remote publication must be called for each remote *DataWriter* that a *DataReader* may discover.

```
struct DDS_PublicationBuiltinTopicData rem_publication_data =
    DDS_PublicationBuiltinTopicData_INITIALIZER;

/* Set Writer's protocol.rtps_object_id */
rem_publication_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 100;

rem_publication_data.topic_name = DDS_String_dup("Example HelloWorld");
rem_publication_data.type_name = DDS_String_dup("HelloWorld");

rem_publication_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

retcode = DPSE_RemotePublication_assert(participant,
                                        "Participant_1",
                                        &rem_publication_data,
                                        HelloWorld_get_key_kind(HelloWorldTypePlugin_
↪get(),
                                        NULL)));
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}
```

Asserting a remote publication requires configuration of all QoS policies necessary to determine matching.

3.8.2 Receiving Samples

Accessing received samples can be done in a few ways:

- **Polling**. Do read or take within a periodic polling loop.
- **Listener**. When a new sample is received, the DataReaderListener's on_data_available is called. Processing is done in the context of the middleware's receive thread. See the above HelloWorldSubscriber_on_data_available callback for example code.
- **Waitset**. Create a waitset, attach it to a status condition with the data_available status enabled, and wait for a received sample to trigger the waitset. Processing is done in the context of the user's application thread. (Note: the code snippet below is taken from the shipped HelloWorld_dpde_waitset example).

```
DDS_WaitSet *waitset = NULL;
struct DDS_Duration_t wait_timeout = { 10, 0 }; /* 10 seconds */
DDS_StatusCondition *dr_condition = NULL;
struct DDS_ConditionSeq active_conditions =
    DDS_SEQUENCE_INITIALIZER(struct DDS_ConditionSeq);

if (!DDS_ConditionSeq_initialize(&active_conditions))
{
    /* failure */
}

if (!DDS_ConditionSeq_set_maximum(&active_conditions, 1))
{
    /* failure */
}

waitset = DDS_WaitSet_new();
if (waitset == NULL )
{
    /* failure */
}

dr_condition = DDS_Entity_get_statuscondition(DDS_DataReader_as_entity(datareader));

retcode = DDS_StatusCondition_set_enabled_statuses(dr_condition,
                                                   DDS_DATA_AVAILABLE_STATUS);
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}

retcode = DDS_WaitSet_attach_condition(waitset,
                                       DDS_StatusCondition_as_condition(dr_condition));
if (retcode != DDS_RETCODE_OK)
{
    /* failure */
```

(continues on next page)

(continued from previous page)

```
}

retcode = DDS_WaitSet_wait(waitset, active_conditions, &wait_timeout);

switch (retcode) {
    case DDS_RETCODE_OK:
    {
        /* This WaitSet only has a single condition attached to it
         * so we can implicitly assume the DataReader's status condition
         * to be active (with the enabled DATA_AVAILABLE status) upon
         * successful return of wait().
         * If more than one conditions were attached to the WaitSet,
         * the returned sequence must be examined using the
         * commented out code instead of the following.
         */

         HelloWorldSubscriber_take_data(HelloWorldDataReader_narrow(datareader));

         /*
         DDS_Long active_len = DDS_ConditionSeq_get_length(&active_conditions);
         for (i = active_len - 1; i >= 0; --i)
         {
             DDS_Condition *active_condition =
                 *DDS_ConditionSeq_get_reference(&active_conditions, i);

             if (active_condition ==
                     DDS_StatusCondition_as_condition(dr_condition))
             {
                 total_samples += HelloWorldSubscriber_take_data(
                                 HelloWorldDataReader_narrow(datareader));
             }
             else if (active_condition == some_other_condition)
             {
                 do_something_else();
             }
         }
         */
         break;
    }
    case DDS_RETCODE_TIMEOUT:
    {
        printf("WaitSet_wait timed out\n");
        break;
    }
    default:
    {
        printf("ERROR in WaitSet_wait: retcode=%d\n", retcode);
        break;
    }
}
```

3.8.3 Filtering Samples

In lieu of supporting Content-Filtered Topics, a DataReaderListener in *Connext DDS Micro* provides callbacks to do application-level filtering per sample.

- **on_before_sample_deserialize**. Through this callback, a received sample is presented to the application before it has been deserialized or stored in the *DataReader*'s queue.
- **on_before_sample_commit**. Through this callback, a received sample is presented to the application after it has been deserialized but before it has been stored in the *DataReader*'s queue.

You control the callbacks' sample_dropped parameter; upon exiting either callback, the *DataReader* will drop the sample if sample_dropped is true. Consequently, dropped samples are not stored in the *DataReader*'s queue and are not available to be read or taken.

Neither callback is associated with a DDS Status. Rather, each is enabled when assigned, to a non-NULL callback.

NOTE: Because it is called after the sample has been deserialized, on_before_sample_commit provides an additional sample_info parameter, containing some of the usual sample information that would be available when the sample is read or taken.

The HelloWorld_dpde example's subscriber has this on_before_sample_commit callback:

```
DDS_Boolean HelloWorldSubscriber_on_before_sample_commit(
    void *listener_data,
    DDS_DataReader *reader,
    const void *const sample,
    const struct DDS_SampleInfo *const sample_info,
    DDS_Boolean *dropped)
{
    HelloWorld *hw_sample = (HelloWorld *)sample;

    /* Drop samples with even-numbered count in msg */
    HelloWorldSubscriber_filter_sample(hw_sample, dropped);

    if (*dropped)
    {
        printf("\nSample filtered, before commit\n\tDROPPED - msg: %s\n",
               hw_sample->msg);
    }

    return DDS_BOOLEAN_TRUE;
}

...

dr_listener.on_before_sample_commit =
    HelloWorldSubscriber_on_before_sample_commit;
```

For more information, see the *Receiving Data* section in the User's Manual.

3.9 Examples

Connext DDS Micro provides buildable example applications, in the **example/** directory of its host bundle. They include a basic HelloWorld application presented in a few different flavors, an RTPS-only emitter, and latency and throughput benchmarking applications.

Each example comes with instructions on how to build and run an application.

All examples are available in C, while the HelloWorld_dpde and HelloWorld_dpde_waitset examples are available in C++.

Note that by the default all the examples link against release libraries. To build release libraries:

```
./resource/scripts/rtime-make --name x64Darwin17clang9.0 --target self --build --config␣
↪Release
```

To build the examples against the debug libraries, specify the the DEBUG option:

```
make DEBUG=Y
```

- **Helloworld_dpse**. Shows how to use rtiddsgen to generate type-support code from a simple HelloWorld IDL-defined type. This examplecreates a publisher and subscriber, and uses dynamic participant, static endpoint discovery to establish communication.
- **HelloWorld_dpde**. Same as the **HelloWorld_dpse** example, except it uses dynamic participant, dynamic endpoint discovery. This example is available in both C and C++.
- **HelloWorld_dpde_waitset**. Same as the HelloWorld_dpde example, except it uses waitsets instead of listener callbacks to access received data.
- **HelloWorld_android**. Example application using Android™ NDK.
- **HelloWorld_static_udp**. Same as HelloWorld_dpde, except it uses static configuration of network interfaces.
- **HelloWorld_transformations**. Same as HelloWorld_dpde, except it uses UDP transformations to send encrypted packets using OpenSSL.
- **RTPS**. Example of an RTPS emitter that bypasses the DDS module and APIs to send RTPS discovery and user data.
- **Latency**. Measures the end-to-end latency of *Connext DDS Micro*.
- **Throughput**. Measures the end-to-end throughput of *Connext DDS Micro*.

Chapter 4

User's Manual

4.1 Data Types

How data is stored or laid out in memory can vary from language to language, compiler to compiler, operating system to operating system, and processor to processor. This combination of language/compiler/operating system/processor is called a *platform*. Any modern middleware must be able to take data from one specific platform (for example, C/gcc.7.3.0/Linux®/PPC) and transparently deliver it to another (for example, C/gcc.7.3.0/Linux/Arm® v8). This process is commonly called *serialization/deserialization*, or *marshalling/demarshalling*.

Connext DDS Micro data samples sent on the same *Connext DDS Micro* topic share a data type. This type defines the fields that exist in the DDS data samples and what their constituent types are. The middleware stores and propagates this meta-information separately from the individual DDS data samples, allowing it to propagate DDS samples efficiently while handling byte ordering and alignment issues for you.

To publish and/or subscribe to data with *Connext DDS Micro*, you will carry out the following steps:

1. Select a type to describe your data and use the *RTI Code Generator* to define a type at compile-time using a language-independent description language.

   The *RTI Code Generator* accepts input in the following formats:

   - **OMG IDL**. This format is a standardized component of the DDS specification. It describes data types with a C++-like syntax. A link to the latest specification can be found here: https://www.omg.org/spec/IDL.
   - **XML in a DDS-specific format**. This XML format is terser, and therefore easier to read and write by hand, than an XSD file. It offers the general benefits of XML-extensibility and ease of integration, while fully supporting DDS-specific data types and concepts. A link to the latest specification, including a description of the XML format, can be found here: https://www.omg.org/spec/DDS-XTypes/.
   - **XSD format**. You can describe data types with XML schemas (XSD). A link to the latest specification, including a description of the XSD format, can be found here: https://www.omg.org/spec/DDS-XTypes/.

Define a type programmatically at run time.

This method may be appropriate for applications with dynamic data description needs: applications for which types change frequently or cannot be known ahead of time.

2. Register your type with a logical name.
3. Create a *Topic* using the type name you previously registered.

   If you've chosen to use a built-in type instead of defining your own, you will use the API constant corresponding to that type's name.

4. Create one or more *DataWriters* to publish your data and one or more *DataReaders* to subscribe to it.

   The concrete types of these objects depend on the concrete data type you've selected, in order to provide you with a measure of type safety.

Whether publishing or subscribing to data, you will need to know how to create and delete DDS data samples and how to get and set their fields. These tasks are described in the section on Working with DDS Data Samples in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access).

4.1.1 Introduction to the Type System

A *user data type* is any custom type that your application defines for use with *RTI Connext DDS Micro*. It may be a structure, a union, a value type, an enumeration, or a typedef (or language equivalents).

Your application can have any number of user data types. They can be composed of any of the primitive data types listed below or of other user data types.

Only structures, unions, and value types may be read and written directly by *Connext DDS Micro*; enums, typedefs, and primitive types must be contained within a structure, union, or value type. In order for a *DataReader* and *DataWriter* to communicate with each other, the data types associated with their respective Topic definitions must be identical.

- octet, char, wchar
- short, unsigned short
- long, unsigned long
- long long, unsigned long long
- float
- double, long double
- boolean
- enum (with or without explicit values)
- bounded string and wstring

The following type-building constructs are also supported:

- module (also called a package or namespace)

- pointer
- array of primitive or user type elements
- bounded sequence of elements—a sequence is a variable-length ordered collection, such as a vector or list
- typedef
- union
- struct
- value type, a complex type that supports inheritance and other object-oriented features

To use a data type with *Connext DDS Micro*, you must define that type in a way the middleware understands and then register the type with the middleware. These steps allow *Connext DDS Micro* to serialize, deserialize, and otherwise operate on specific types. They will be described in detail in the following sections.

Sequences

A sequence contains an ordered collection of elements that are all of the same type. The operations supported in the sequence are documented in the C API Reference and C++ API Reference HTML documentation.

Elements in a sequence are accessed with their index, just like elements in an array. Indices start at zero in all APIs. Unlike arrays, however, sequences can grow in size. A sequence has two sizes associated with it: a physical size (the "maximum") and a logical size (the "length"). The physical size indicates how many elements are currently allocated by the sequence to hold; the logical size indicates how many valid elements the sequence actually holds. The length can vary from zero up to the maximum. Elements cannot be accessed at indices beyond the current length.

A sequence must be declared as bounded. A sequence's "bound" is the maximum number of elements that the sequence can contain at any one time. A finite bound is very important because it allows *RTI Connext DDS Micro* to preallocate buffers to hold serialized and deserialized samples of your types; these buffers are used when communicating with other nodes in your distributed system.

By default, any unbounded sequences found in an IDL file will be given a default bound of 100 elements. This default value can be overwritten using *RTI Code Generator*'s **-sequenceSize** command-line argument (see the Command-Line Arguments chapter in the *RTI Code Generator User's Manual*, available here if you have Internet access).

Strings and Wide Strings

Connext DDS Micro supports both strings consisting of single-byte characters (the IDL string type) and strings consisting of wide characters (IDL wstring). The wide characters supported by *Connext DDS Micro* are large enough to store 4-byte Unicode/UTF16 characters.

Like sequences, strings must be bounded. A string's "bound" is its maximum length (not counting the trailing NULL character in C and C++).

In C and Traditional C++, strings are mapped to char*. Optionally, the mapping in Traditional C++ can be changed to **std::string** by generating code with the option **-useStdString**.

By default, any unbounded string found in an IDL file will be given a default bound of 255 elements. This default value can be overwritten using *RTI Code Generator's* **-stringSize** command-line argument (see the Command-Line Arguments chapter in the *RTI Code Generator User's Manual*, available here if you have Internet access).

IDL String Encoding

The "Extensible and Dynamic Topic Types for DDS specification" (https://www.omg.org/spec/DDS-XTypes/) standardizes the default encoding for strings to UTF-8. This encoding shall be used as the wire format. Language bindings may use the representation that is most natural in that particular language. If this representation is different from UTF-8, the language binding shall manage the transformation to/from the UTF-8 wire representation.

As an extension, *Connext DDS Micro* offers ISO_8859_1 as an alternative string wire encoding.

This section describes the encoding for IDL strings across different languages in *Connext DDS Micro* and how to configure that encoding.

- C, Traditional C++

  IDL strings are mapped to a NULL-terminated array of DDS_Char (char*). Users are responsible for using the right character encoding (UTF-8 or ISO_8859_1) when populating the string values. This applies to all generated code, DynamicData, and Built-in data types. The middleware does not transform from the language binding encoding to the wire encoding.

IDL Wide Strings Encoding

The "Extensible and Dynamic Topic Types for DDS specification" (https://www.omg.org/spec/DDS-XTypes/) standardizes the default encoding for wide strings to UTF-32. This encoding shall be used as the wire format.

Wide-string characters have a size of 4 bytes on the wire with UTF-32 encoding.

Language bindings may use the representation that is most natural in that particular language. If this representation is different from UTF-32, the language binding shall manage the transformation to/from the UTF-32 wire representation.

- C, Traditional C++

  IDL wide strings are mapped to a NULL-terminated array of DDS_Wchar (DDS_Wchar*). DDS_WChar is an unsigned 4-byte integer. Users are responsible for using the right character encoding (UTF-32) when populating the wide-string values. This applies to all generated code, DynamicData, and Built-in data types. *Connext DDS Micro* does not transform from the language binding encoding to the wire encoding.

Sending Type Information on the Network

Connext DDS Micro can send type information the network using a concept called type objects. A type objects is a description of a type suitable to network transmission, and is commonly used by for example tools to visualize data from any application.

However, please note that *Connext DDS Micro* does not support sending type information on the network. Instead, tools can load type information from XML files generated from IDL using *rtiddsgen*. Please refer to the *RTI Code Generator's User's Manual* for more information (available here if you have Internet access).

4.1.2 Creating User Data Types with IDL

You can create user data types in a text file using IDL (Interface Description Language). IDL is programming-language independent, so the same file can be used to generate code in C and Traditional C++. *RTI Code Generator* parses the IDL file and automatically generates all the necessary routines and wrapper functions to bind the types for use by *Connext DDS Micro* at run time. You will end up with a set of required routines and structures that your application and *Connext DDS Micro* will use to manipulate the data.

Please refer to the section on Creating User Data Types with IDL in the *RTI Connext DDS Core Libraries User's Manual* for more information (available here if you have Internet access).

Note: Not all features in *RTI Code Generator* are supported when generating code for *Connext DDS Micro*, see *Unsupported Features of rtiddsgen with Connext DDS Micro*.

4.1.3 Working with DDS Data Samples

You should now understand how to define and work with data types. Now that you have chosen one or more data types to work with, this section will help you understand how to create and manipulate objects of those types.

In C:

You create and delete your own objects from factories, just as you create *Connext DDS Micro* objects from factories. In the case of user data types, the factory is a singleton object called the type support. Objects allocated from these factories are deeply allocated and fully initialized.

```
/* In the generated header file: */
struct MyData {
    char* myString;
};
/* In your code: */
MyData* sample = MyDataTypeSupport_create_data();
char* str = sample->myString; /*empty, non-NULL string*/
/* ... */
MyDataTypeSupport_delete_data(sample);
```

In Traditional C++:

Without the **-constructor option**, you create and delete objects using the TypeSupport factories.

```
MyData* sample = MyDataTypeSupport::create_data();
char* str = sample->myString; // empty, non-NULL string
// ...
MyDataTypeSupport::delete_data(sample);
```

Please refer to the section on Working with DDS Data Samples in the *RTI Connext DDS Core Libraries User's Manual* for more information (available here if you have Internet access).

4.2 DDS Entities

The main classes extend an abstract base class called a DDS *Entity*. Every DDS *Entity* has a set of associated events known as statuses and a set of associated Quality of Service Policies (QosPolicies). In addition, a *Listener* may be registered with the *Entity* to be called when status changes occur. DDS *Entities* may also have attached DDS *Conditions*, which provide a way to wait for status changes. *Figure 4.1: Overview of DDS Entities* presents an overview in a UML diagram.

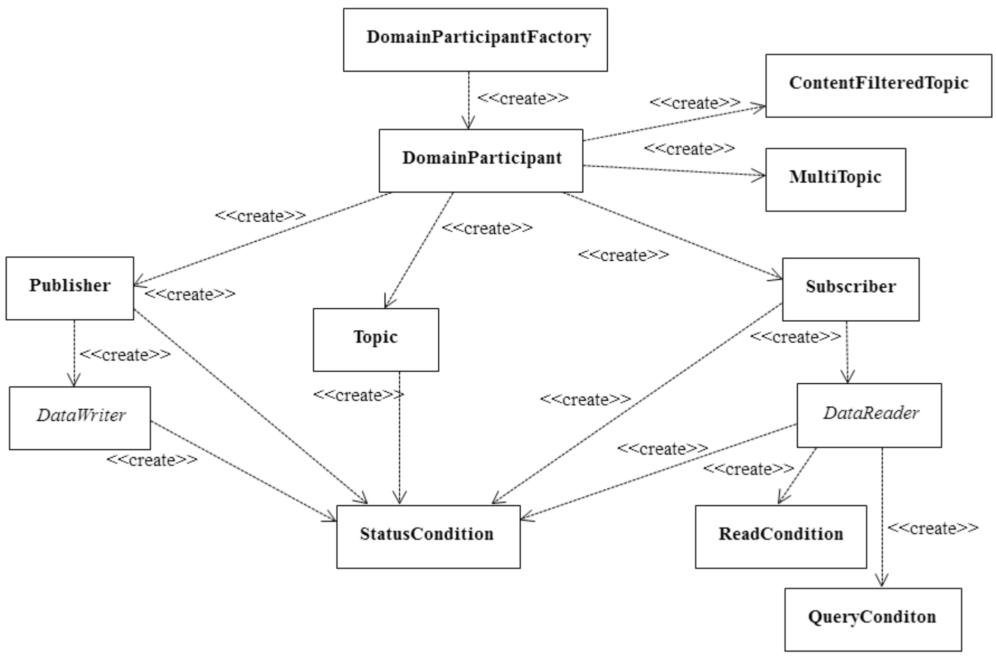


Figure 4.1: Overview of DDS Entities

Please note that *RTI Connext DDS Micro* does not support the following:

- **MultiTopic**
- **ContentFileteredTopic**
- **ReadCondition**
- **QueryConditions**

For a general description of DDS *Entities* and their operations, please refer to the DDS Entities chapter in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access). Note that *RTI Connext DDS Micro* does not support all APIs and QosPolicies; please refer to the C API Reference and C++ API Reference documentation for more information.

4.3 Sending Data

This section discusses how to create, configure, and use *Publishers* and *DataWriters* to send data. It describes how these *Entities* interact, as well as the types of operations that are available for them.

The goal of this section is to help you become familiar with the *Entities* you need for sending data. For up-to-date details such as formal parameters and return codes on any mentioned operations, please see the C API Reference and C++ API Reference documentation.

4.3.1 Preview: Steps to Sending Data

To send DDS samples of a data instance:

1. Create and configure the required *Entities*:
   (a) Create a *DomainParticipant*.
   (b) Register user data types with the *DomainParticipant*. For example, the '**FooDataType**'.
   (c) Use the *DomainParticipant* to create a *Topic* with the registered data type.
   (d) Use the *DomainParticipant* to create a *Publisher*.
   (e) Use the *Publisher* or *DomainParticipant* to create a *DataWriter* for the *Topic*.
   (f) Use a type-safe method to cast the generic *DataWriter* created by the *Publisher* to a type-specific *DataWriter*. For example, '**FooDataWriter**'. Optionally, register data instances with the *DataWriter*. If the *Topic*'s user data type contain key fields, then registering a data instance (data with a specific key value) will improve performance when repeatedly sending data with the same key. You may register many different data instances; each registration will return an instance handle corresponding to the specific key value. For non-keyed data types, instance registration has no effect.
2. Every time there is changed data to be published:
   (a) Store the data in a variable of the correct data type (for instance, variable '**Foo**' of the type '**FooDataType**').
   (b) Call the **FooDataWriter**'s **write()** operation, passing it a reference to the variable '**Foo**'.
      - For non-keyed data types or for non-registered instances, also pass in **DDS_HANDLE_NIL**.
      - For keyed data types, pass in the instance handle corresponding to the instance stored in 'Foo', if you have registered the instance previously. This means that the data stored in 'Foo' has the same key value that was used to create instance handle.

   (c) The **write()** function will take a snapshot of the contents of '**Foo**' and store it in *Connext DDS* internal buffers from where the DDS data sample is sent under the criteria set by the *Publisher's* and *DataWriter's* QosPolicies. If there are matched *DataReaders*, then the DDS data sample will have been passed to the physical transport plug-in/device driver by the time that **write()** returns.

4.3.2 Publishers

An application that intends to publish information needs the following *Entities*: *DomainParticipant*, *Topic*, *Publisher*, and *DataWriter*. All *Entities* have a corresponding specialized *Listener* and a set of QosPolicies. A *Listener* is how *Connext DDS* notifies your application of status changes relevant to the *Entity*. The QosPolicies allow your application to configure the behavior and resources of the *Entity*.

- A *DomainParticipant* defines the DDS domain in which the information will be made available.
- A *Topic* defines the name under which the data will be published, as well as the type (format) of the data itself.
- An application writes data using a *DataWriter*. The *DataWriter* is bound at creation time to a *Topic*, thus specifying the name under which the *DataWriter* will publish the data and the type associated with the data. The application uses the *DataWriter's* **write()** operation to indicate that a new value of the data is available for dissemination.
- A *Publisher* manages the activities of several *DataWriters*. The *Publisher* determines when the data is actually sent to other applications. Depending on the settings of various QosPolicies of the *Publisher* and *DataWriter*, data may be buffered to be sent with the data of other *DataWriters* or not sent at all. By default, the data is sent as soon as the *DataWriter's* **write()** function is called.

  You may have multiple *Publishers*, each managing a different set of *DataWriters*, or you may choose to use one *Publisher* for all your *DataWriters*.

4.3.3 DataWriters

To create a *DataWriter*, you need a *DomainParticipant*, *Publisher*, and a *Topic*.

You need a *DataWriter* for each *Topic* that you want to publish. For more details on all operations, see the C API Reference and C++ API Reference documentation.

For more details on creating, deleting, and setting up *DataWriters*, see the DataWriters section in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access).

4.3.4 Publisher/Subscriber QosPolicies

Please refer to the C API Reference and C++ API Reference for details on supported QosPolicies.

4.3.5 DataWriter QosPolicies

Please refer to the C API Reference and C++ API Reference for details on supported QosPolicies.

4.4 Receiving Data

This section discusses how to create, configure, and use *Subscribers* and *DataReaders* to receive data. It describes how these objects interact, as well as the types of operations that are available for them.

The goal of this section is to help you become familiar with the *Entities* you need for receiving data. For up-to-date details such as formal parameters and return codes on any mentioned operations, please see the C API Reference and C++ API Reference documentation.

4.4.1 Preview: Steps to Receiving Data

There are three ways to receive data:

- Your application can explicitly check for new data by calling a *DataReader's* **read()** or **take()** operation. This method is also known as *polling for data*.
- Your application can be notified asynchronously whenever new DDS data samples arrive—this is done with a *Listener* on either the *Subscriber* or the *DataReader*. *RTI Connext DDS Micro* will invoke the *Listener's* callback routine when there is new data. Within the callback routine, user code can access the data by calling **read()** or **take()** on the *DataReader*. This method is the way for your application to receive data with the least amount of latency.
- Your application can wait for new data by using *Conditions* and a *WaitSet*, then calling **wait()**. *Connext DDS Micro* will block your application's thread until the criteria (such as the arrival of DDS samples, or a specific status) set in the *Condition* becomes true. Then your application resumes and can access the data with **read()** or **take()**.

The *DataReader's* **read()** operation gives your application a copy of the data and leaves the data in the *DataReader's* receive queue. The *DataReader's* **take()** operation removes data from the receive queue before giving it to your application.

To prepare to receive data, create and configure the required Entities:

1. Create a *DomainParticipant*.
2. Register user data types with the *DomainParticipant*. For example, the '**FooDataType**'.
3. Use the *DomainParticipant* to create a *Topic* with the registered data type.
4. Use the *DomainParticipant* to create a *Subscriber*.
5. Use the *Subscriber* or *DomainParticipant* to create a *DataReader* for the *Topic*.
6. Use a type-safe method to cast the generic *DataReader* created by the *Subscriber* to a type-specific *DataReader*. For example, '**FooDataReader**'.

Then use one of the following mechanisms to receive data.

- To receive DDS data samples by polling for new data:
  - Using a **FooDataReader**, use the **read()** or **take()** operations to access the DDS data samples that have been received and stored for the *DataReader*. These operations can be invoked at any time, even if the receive queue is empty.
- To receive DDS data samples asynchronously:
  - Install a *Listener* on the *DataReader* or *Subscriber* that will be called back by an internal *Connext DDS Micro* thread when new DDS data samples arrive for the *DataReader*.

1. Create a *DDSDataReaderListener* for the *FooDataReader* or a *DDSSubscriberListener* for *Subscriber*. In C++ you must derive your own *Listener* class from those base classes. In C, you must create the individual functions and store them in a structure.

   If you created a *DDSDataReaderListener* with the **on_data_available()** callback enabled: **on_data_available()** will be called when new data arrives for that **DataReader**.

   If you created a *DDSSubscriberListener* with the **on_data_on_readers()** callback enabled: **on_data_on_readers()** will be called when data arrives for any *DataReader* created by the *Subscriber*.

2. Install the *Listener* on either the *FooDataReader* or *Subscriber*.

   For the *DataReader*, the *Listener* should be installed to handle changes in the **DATA_AVAILABLE** status.

   For the *Subscriber*, the *Listener* should be installed to handle changes in the **DATA_ON_READERS** status.

3. Only 1 *Listener* will be called back when new data arrives for a *DataReader*.

   Connext DDS Micro will call the *Subscriber's Listener* if it is installed. Otherwise, the *DataReader's Listener* is called if it is installed. That is, the **on_data_on_readers()** operation takes precedence over the **on_data_available()** operation.

   If neither *Listeners* are installed or neither *Listeners* are enabled to handle their respective statuses, then *Connext DDS Micro* will not call any user functions when new data arrives for the *DataReader*.

4. In the **on_data_available()** method of the *DDSDataReaderListener*, invoke **read()** or **take()** on the *FooDataReader* to access the data.

   If the **on_data_on_readers()** method of the *DDSSubscriberListener* is called, the code can invoke **read()** or **take()** directly on the *Subscriber's DataReaders* that have received new data. Alternatively, the code can invoke the *Subscriber's* **notify_datareaders()** operation. This will in turn call the **on_data_available()** methods of the *DataReaderListeners* (if installed and enabled) for each of the *DataReaders* that have received new DDS data samples.

To wait (block) until DDS data samples arrive:

1. Use the *DataReader* to create a *StatusCondition* that describes the DDS samples for which you want to wait. For example, you can specify that you want to wait for never-before-seen DDS samples from *DataReaders* that are still considered to be 'alive.'
2. Create a *WaitSet*.
3. Attach the *StatusCondition* to the *WaitSet*.
4. Call the *WaitSet's* **wait()** operation, specifying how long you are willing to wait for the desired DDS samples. When **wait()** returns, it will indicate that it timed out, or that the attached Condition become true (and therefore the desired DDS samples are available).
5. Using a **FooDataReader**, use the **read()** or **take()** operations to access the DDS data samples that have been received and stored for the *DataReader*.

4.4.2 Subscribers

An application that intends to subscribe to information needs the following *Entities*: *DomainParticipant*, *Topic*, *Subscriber*, and *DataReader*. All *Entities* have a corresponding specialized *Listener* and a set of QosPolicies. The *Listener* is how *RTI Connext DDS Micro* notifies your application of status changes relevant to the *Entity*. The QosPolicies allow your application to configure the behavior and resources of the *Entity*.

- The *DomainParticipant* defines the DDS domain on which the information will be available.
- The *Topic* defines the name of the data to be subscribed, as well as the type (format) of the data itself.
- The *DataReader* is the *Entity* used by the application to subscribe to updated values of the data. The *DataReader* is bound at creation time to a *Topic*, thus specifying the named and typed data stream to which it is subscribed. The application uses the *DataWriter's* **read()** or **take()** operation to access DDS data samples received for the *Topic*.
- The *Subscriber* manages the activities of several *DataReader* entities. The application receives data using a *DataReader* that belongs to a *Subscriber*. However, the *Subscriber* will determine when the data received from applications is actually available for access through the *DataReader*. Depending on the settings of various QosPolicies of the *Subscriber* and *DataReader*, data may be buffered until DDS data samples for associated *DataReaders* are also received. By default, the data is available to the application as soon as it is received.

For more information on creating and deleting *Subscribers*, as well as setting QosPolicies, see the Subscribers section in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access).

4.4.3 DataReaders

To create a *DataReader*, you need a *DomainParticipant*, a *Topic*, and a *Subscriber*. You need at least one *DataReader* for each *Topic* whose DDS data samples you want to receive.

For more details on all operations, see the C API Reference and C++ API Reference HTML documentation.

4.4.4 Using DataReaders to Access Data (Read & Take)

For user applications to access the data received for a *DataReader*, they must use the type-specific derived class or set of functions in the C API Reference. Thus for a user data type '**Foo**', you must use methods of the **FooDataReader** class. The type-specific class or functions are automatically generated if you use *RTI Code Generator*.

4.4.5 Subscriber QosPolicies

Please refer to the C API Reference and C++ API Reference for details on supported QosPolicies.

4.4.6 DataReader QosPolicies

Please refer to the C API Reference and C++ API Reference for details on supported QosPolicies.

4.5 DDS Domains

This section discusses how to use *DomainParticipants*. It describes the types of operations that are available for them and their QosPolicies.

The goal of this section is to help you become familiar with the objects you need for setting up your *RTI Connext DDS Micro* application. For specific details on any mentioned operations, see the C API Reference and C++ API Reference documentation.

4.5.1 Fundamentals of DDS Domains and DomainParticipants

DomainParticipants are the focal point for creating, destroying, and managing other *RTI Connext DDS Micro* objects. A DDS *domain* is a logical network of applications: only applications that belong to the same DDS *domain* may communicate using *Connext DDS Micro*. A DDS *domain* is identified by a unique integer value known as a domain ID. An application participates in a DDS domain by creating a *DomainParticipant* for that domain ID.

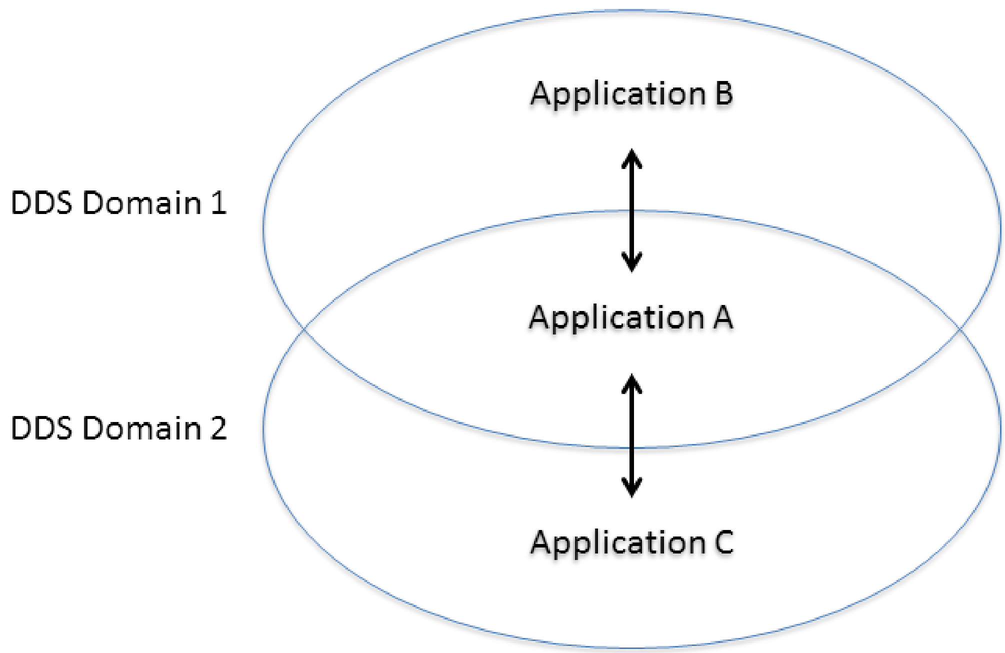


Figure 4.2: Relationship between Applications and DDS Domains
Applications can belong to multiple DDS domains—*A* belongs to DDS domains 1 and 2. Applications in the same DDS domain can communicate with each other, such as *A* and *B*, or *A* and *C*. Applications in different DDS domains, such as *B* and *C*, are not even aware of each other and will not exchange messages.

As seen in *Figure 4.2: Relationship between Applications and DDS Domains*, a single application can participate in multiple DDS domains by creating multiple *DomainParticipants* with different domain IDs. *DomainParticipants* in the same DDS domain form a logical network; they are isolated from *DomainParticipants* of other DDS domains, even those running on the same set of physical computers sharing the same physical network. *DomainParticipants* in different DDS domains will never exchange messages with each other. Thus, a DDS domain establishes a "virtual network" linking all *DomainParticipants* that share the same domain ID.

An application that wants to participate in a certain DDS domain will need to create a *DomainParticipant*. As seen in *Figure 4.3: DDS Domain Module*, a *DomainParticipant* object is a container for all other *Entities* that belong to the same DDS domain. It acts as factory for the *Publisher*, *Subscriber*, and *Topic* entities. (As seen in *Sending Data* and *Receiving Data*, in turn, *Publishers* are factories for *DataWriters* and *Subscribers* are factories for *DataReaders*.) *DomainParticipants* cannot contain other *DomainParticipants*.

Like all *Entities*, *DomainParticipants* have QosPolicies and *Listeners*. The *DomainParticipant* entity also allows you to set 'default' values for the QosPolicies for all the entities created from it or from the entities that it creates (*Publishers*, *Subscribers*, *Topics*, *DataWriters*, and *DataReaders*).

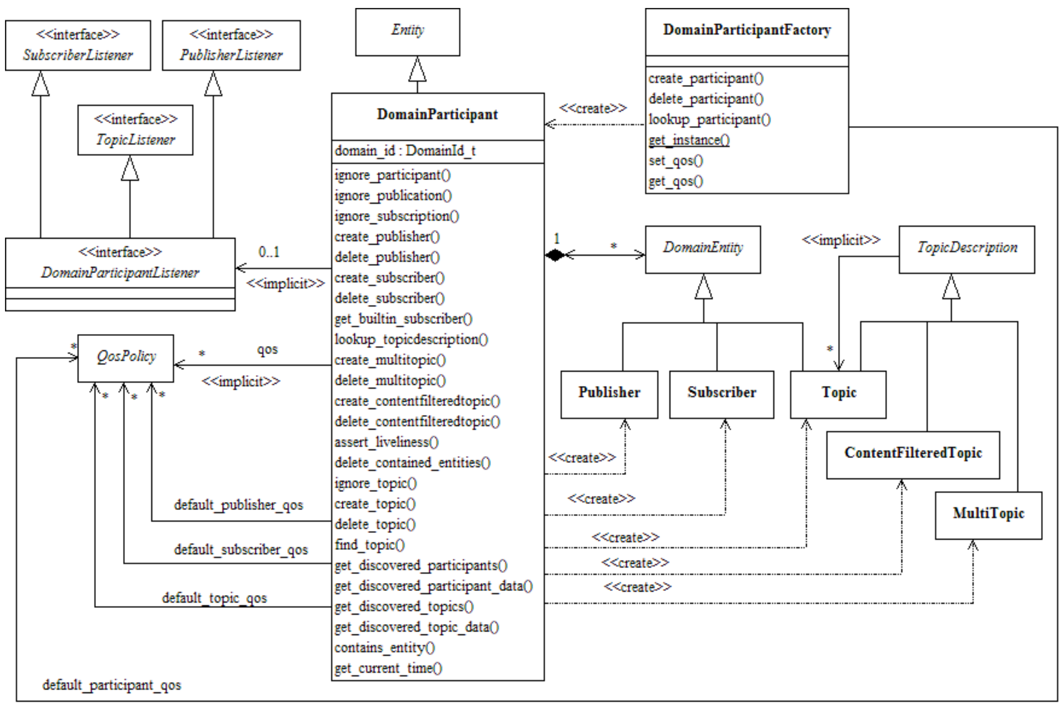


Figure 4.3: DDS Domain Module
Note: MultiTopics are not supported.

4.5.2 Discovery Announcements

Each *DomainParticipant* announces information about itself, such as which locators other *DomainParticipants* must use to communicate with it. A locator is an address that consists of an address kind, a port number, and an address. Four locator types are defined:

- A **unicast meta-traffic locator**. This locator type is used to identify where unicast discovery messages shall be sent. A maximum of four locators of this type can be specified.
- A **multicast meta-traffic locator**. This locator type is used to identify where multicast discovery messages shall be sent. A maximum of four locators of this type can be specified.
- A **unicast user-traffic locator**. This locator type is used to identify where unicast user-traffic messages shall be sent. A maximum of four locators of this type can be specified.

- A **multicast user-traffic locator**. This locator type is used to identify where multicast user-traffic messages shall be sent. A maximum of four locators of this type can be specified.

It is important to note that a maximum of *four* locators of *each* kind can be sent in a *DomainParticipant* discovery message.

The locators in a *DomainParticipant*'s discovery announcement is used for two purposes:

- It informs other *DomainParticipants* where to send their discovery announcements to this *DomainParticipants*.
- It informs the *DataReaders* and *DataWriters* in other *DomainParticipants* where to send data to the *DataReaders* and *DataWriters* in this *DomainParticipant* unless a *DataReader* or *DataWriter* specifies its own locators.

If a *DataReader* or *DataWriter* specifies their own locators, only user-traffic locators can be specified, then the exact same rules apply as for the *DomainParticipant*.

This document uses *address* and *locator* interchangeably. An address corresponds to the port and address part of a locator. The same address may exist as different kinds, in which case they are unique.

For more details about the discovery process, see the *Discovery* section.

4.6 Transports

4.6.1 Introduction

RTI Connext DDS Micro has a pluggable-transports architecture. The core of *Connext DDS Micro* is transport agnostic—it does not make any assumptions about the actual transports used to send and receive messages. Instead, *Connext DDS Micro* uses an abstract "transport API" to interact with the transport plugins that implement that API. A transport plugin implements the abstract transport API, and performs the actual work of sending and receiving messages over a physical transport.

In *Connext DDS Micro* a Network Input/Output (NETIO) interface is a software layer that may send and/or receive data from a higher and/or lower level locally, as well as communicate with a peer. A transport is a NETIO interface that is at the lowest level of the protocol stack. For example, the UDP NETIO interface is a transport.

A transport can send and receive on addresses as defined by the concrete transport. For example, the *Connext DDS Micro* UDP transport can listen to and send to UDPv4 ports and addresses. In order to establish communication between two transports, the addresses that the transport can listen to must be determined and announced to other *DomainParticipants* that want to communicate with it. This document describes how the addresses are reserved and how these addresses are used by the DDS layer in *Connext DDS Micro*.

While the NETIO interface is not limited to DDS, the rest of this document is written in the context of how *Connext DDS Micro* uses the NETIO interfaces as part of the DDS implementation.

Note that *Connext DDS Micro* 2 does not support RTPS fragmentation and is limited to data types less than or equal to 63000 bytes.

4.6.2 Transport Registration

RTI Connext DDS Micro supports different transports and transports must be registered with *RTI Connext DDS Micro* before they can be used. A transport must be given a name when it is registered and this name is later used when configuring discovery and user-traffic. A transport name cannot exceed 7 UTF-8 characters.

The following example registers the UDP transport with *RTI Connext DDS Micro* and makes it available to all DDS applications within the same memory space. Please note that each DDS applications creates its *own* instance of a transport. Resources are *not* shared between instances of a transport unless stated.

For example, to register two UDP transports with the names myudp1 and myudp2, the following code is required:

```
DDS_DomainParticipantFactory *factory;
RT_Registry_T *registry;
struct UDP_InterfaceFactoryProperty udp_property;

factory = DDS_DomainParticipantFactory_get_instance();
registry = DDS_DomainParticipantFactory_get_registry(factory);

/* Set UDP properties */
if (!RT_Registry_register(registry,"myudp1",
                          UDP_InterfaceFactory_get_interface(),
                          &udp_property._parent._parent,NULL))
{
    return error;
}

/* Set UDP properties */
if (!RT_Registry_register(registry,"myudp2",
                          UDP_InterfaceFactory_get_interface(),
                          &udp_property._parent._parent,NULL))
{
    return error;
}
```

Before a DomainParticipant can make use of a registered transport, it must enable it for use within the DomainParticipant. This is done by setting the TransportQoS. For example, to enable only myudp1, the following code is required (error checking is not shown for clarity):

```
DDS_StringSeq_set_maximum(&dp_qos.transports.enabled_transports,1);
DDS_StringSeq_set_length(&dp_qos.transports.enabled_transports,1);
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports,0) =
                                                  REDA_String_dup("myudp1");
```

To enable both transports:

```
DDS_StringSeq_set_maximum(&dp_qos.transports.enabled_transports,2);
DDS_StringSeq_set_length(&dp_qos.transports.enabled_transports,2);
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports,0) =
```

(continues on next page)

(continued from previous page)

```
                                                    REDA_String_dup("myudp1");
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports,0) =
                                                    REDA_String_dup("myudp2");
```

Before enabled transports may be used for communication in *Connext DDS Micro*, they must be registered and added to the DiscoveryQos and UserTrafficQos policies. Please see the section on *Discovery* for details.

4.6.3 Transport Addresses

Address reservation is the process to determine which locators should be used in the discovery announcement. Which transports and addresses to be used is determined as described in *Discovery*.

When a *DomainParticipant* is created, it calculates a port number and tries to reserve this port on all addresses available in *all* the transports based on the registration properties. If the port cannot be reserved on all transports, then it release the port on *all* transports and tries again. If no free port can be found the process fails and the *DomainParticipant* cannot be created.

The number of locators which can be announced is limited to *only* the first *four* for each type across *all* transports available for each policy. If more than four are available of any kind, these are *ignored*. This is by design, although it may be changed in the future. The order in which the locators are read is also not known, thus the four locators which will be used are not deterministic.

To ensure that *all* the desired addresses and *only* the desired address are used in a transport, follow these rules:

- Make sure that no more than four unicast addresses and four multicast addresses can be returned across *all* transports for discovery traffic.
- Make sure that no more than four unicast addresses and four multicast addresses can be returned across *all* transports for user traffic.
- Make sure that no more than four unicast addresses and four multicast addresses can be returned across *all* transports for user-traffic, for *DataReader* and *DataWriter* specific locators, and that they do *not* duplicate any of the *DomainParticipant*'s locators.

4.6.4 Transport Port Number

The port number of a locator is not directly configurable. Rather, it is configured indirectly by the DDS_WireProtocolQosPolicy (rtps_well_known_ports) of the *DomainParticipant's* QoS, where a well-known, interoperable RTPS port number is assigned.

4.6.5 INTRA Transport

The builtin intra participant transport (INTRA) is a transport that bypasses RTPS and reduces the number of data-copies from three to one for data published by a *DataWriter* to a *DataReader* within the same participant. When a sample is published, it is copied directly to the data reader's cache (if there is space). This transport is used for communication between *DataReaders* and *DataWriters* created within the same participant by default.

Please refer to *Threading Model* for important details regarding application constraints when using this transport.

Registering the INTRA Transport

The builtin INTRA transport is a *RTI Connext DDS Micro* component that is automatically registered when the DDS_DomainParticipantFactory_get_instance() method is called. By default, data published by a *DataWriter* is sent to all *DataReaders* within the same participant using the INTRA transport.

In order to prevent the INTRA transport from being used it is necessary to remove it as a transport and a user-data transport. The following code shows how to only use the builtin UDP transport for user-data.

```
struct DDS_DomainParticipantQos dp_qos =
                            DDS_DomainParticipantQos_INITIALIZER;

REDA_StringSeq_set_maximum(&dp_qos.transports.enabled_transports,1);
REDA_StringSeq_set_length(&dp_qos.transports.enabled_transports,1);
*REDA_StringSeq_get_reference(&dp_qos.transports.enabled_transports,0) =
                                    REDA_String_dup(NETIO_DEFAULT_UDP_NAME);

/* Use only unicast for user-data traffic. */
REDA_StringSeq_set_maximum(&dp_qos.user_traffic.enabled_transports,1);
REDA_StringSeq_set_length(&dp_qos.user_traffic.enabled_transports,1);
*REDA_StringSeq_get_reference(&dp_qos.user_traffic.enabled_transports,0) =
                                    REDA_String_dup("_udp://");
```

Note that the INTRA transport is never used for discovery traffic internally. It is not possible to disable matching of *DataReaders* and *DataWriters* within the same participant.

Reliability and Durability

Because a sample sent over INTRA bypasses the RTPS reliability and DDS durability queue, the Reliability and Durability Qos policies are *not* supported by the INTRA transport. However, by creating all the *DataReaders* before the *DataWriters* durability is not required.

Threading Model

The INTRA transport does not create any threads. Instead, a *DataReader* receives data over the INTRA transport in the context of the *DataWriter*'s *send thread*.

This model has two *important limitations*:

- Because a *DataReader*'s on_data_available() listener is called in the context of the *DataWriter*'s send thread, a *DataReader* may potentially process data at a different priority than intended (the *DataWriter*'s). While it is generally not recommended to process data in a *DataReader*'s on_data_available() listener, it is particularly important *to not do so* when using the INTRA transport. Instead, use a DDS WaitSet or a similar construct to wake up a separate thread to process data.
- Because a *DataReader*'s on_data_available() listener is called in the context of the *DataWriter*'s send thread, any method called in the on_data_available() listener is done

in the context of the *DataWriter*'s stack. Calling a *DataWriter* **write()** in the callback could result in an infinite call stack. Thus, it is recommended *not* to call in this listener any *Connext DDS Micro* APIs that write data.

4.6.6 UDP Transport

This section describes the builtin *RTI Connext DDS Micro* UDP transport and how to configure it.

The builtin UDP transport (UDP) is a fairly generic UDPv4 transport. *Connext DDS Micro* supports the following functionality:

- Unicast
- Multicast
- Automatic detection of available network interfaces
- Manual configuration of network interfaces
- Allow/Deny lists to select which network interfaces can be used
- Simple NAT configuration
- Configuration of receive threads

Registering the UDP Transport

The builtin UDP transport is a *Connext DDS Micro* component that is automatically registered when the DDS_DomainParticipantFactory_get_instance() method is called. To change the UDP configuration, it is necessary to first unregister the transport as shown below:

```
DDS_DomainParticipantFactory *factory = NULL;
RT_Registry_T *registry = NULL;

factory = DDS_DomainParticipantFactory_get_instance();
registry = DDS_DomainParticipantFactory_get_registry(factory);

/* The builtin transport does not return any properties (3rd param) or
 * listener (4th param)
 */
if (!RT_Registry_unregister(registry, "_udp", NULL, NULL))
{
    /* ERROR */
}
```

When a component is registered, the registration takes the properties and a listener as the 3rd and 4th parameters. In general, it is up to the caller to manage the memory for the properties and the listeners. There is no guarantee that a component makes a copy.

The following code-snippet shows how to register the UDP transport with new parameters.

```
struct UDP_InterfaceFactoryProperty *udp_property = NULL;
```

(continues on next page)

(continued from previous page)

```
/* Allocate a property structure for the heap, it must be valid as long
 * as the component is registered
 */
udp_property = (struct UDP_InterfaceFactoryProperty *)
                    malloc(sizeof(struct UDP_InterfaceFactoryProperty));
if (udp_property != NULL)
{
    *udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    /* Only allow network interface "eth0" to be used;
     */
    REDA_StringSeq_set_maximum(&udp_property->allow_interface, 1);
    REDA_StringSeq_set_length(&udp_property->allow_interface, 1);

    *REDA_StringSeq_get_reference(&udp_property->allow_interface, 0) =
                                            REDA_String_dup("eth0");

    /* Register the transport again, using the builtin name
     */
    if (!RT_Registry_register(registry, "_udp",
                         UDP_InterfaceFactory_get_interface(),
                        (struct RT_ComponentFactoryProperty*)udp_property,
                        NULL))
    {
        /* ERROR */
    }
}
else
{
    /* ERROR */
}
```

It should be noted that the UDP transport can be registered with any name, but all transport QoS policies and initial peers must refer to this name. If a transport is referred to and it does not exist, an error message is logged.

It is possible to register multiple UDP transports with a DomainParticipantFactory. It is also possible to use different UDP transports within the same *DomainParticipant* when multiple network interfaces are available (either physical or virtual).

When UDP transformations are enabled, this feature is always enabled and determined by the allow_interface and deny_interface lists. If any of the lists are non-empty the UDP transports will bind each receive socket to the specific interfaces.

When UDP transformations are not enabled, this feature is determined by the value of the enable_interface_bind. If this value is set to **RTI_TRUE** and the allow_interface and/or deny_interface properties are non-empty, the receive sockets are bound to specific interfaces.

Threading Model

The UDP transport creates one receive thread for each unique UDP receive address and port. Thus, by default, three UDP threads are created:

- A multicast receive thread for discovery data (assuming multicast is available and enabled)
- A unicast receive thread for discovery data
- A unicast receive thread for user data

Additional threads may be created depending on the transport configuration for a *DomainParticipant*, *DataReader*, and *DataWriter*. The UDP transport creates threads based on the following criteria:

- Each unique unicast port creates a new thread
- Each unique multicast address *and* port creates a new thread

For example, if a *DataReader* specifies its own multicast receive address, a new receive thread will be created.

Configuring UDP Receive Threads

All threads in the UDP transport share the same thread settings. It is important to note that all the UDP properties must be set before the UDP transport is registered. *Connext DDS Micro* preregisters the UDP transport with default settings when the DomainParticipantFactory is initialized. To change the UDP thread settings, use the following code.

```
struct UDP_InterfaceFactoryProperty *udp_property = NULL;
struct UDP_InterfaceFactoryProperty udp_property =
                            UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

/* Allocate a property structure for the heap, it must be valid as long
 * as the component is registered
 */
udp_property = (struct UDP_InterfaceFactoryProperty *)
                    malloc(sizeof(struct UDP_InterfaceFactoryProperty));
*udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

/* Please refer to OSAPI_ThreadOptions for possible options */
udp_property->recv_thread.options = ...;

/* The stack-size is platform dependent, it is passed directly to the OS */
udp_property->recv_thread.stack_size = ....

/* The priority is platform dependent, it is passed directly to the OS */
udp_property->recv_thread.priority = ....

if (!RT_Registry_register(registry, "_udp",
                          UDP_InterfaceFactory_get_interface(),
                          (struct RT_ComponentFactoryProperty*)udp_property,
                          NULL))
{
    /* ERROR */
}
```

UDP Configuration

All the configuration of the UDP transport is done via the UDP_InterfaceFactoryProperty.

```
struct UDP_InterfaceFactoryProperty
{
    /* Inherited from */
    struct NETIO_InterfaceFactoryProperty _parent;

    /* Sequence of allowed interface names */
    struct REDA_StringSeq allow_interface;

    /* Sequence of denied interface names */
    struct REDA_StringSeq deny_interface;

    /* The size of the send socket buffer */
    RTI_INT32 max_send_buffer_size;

    /* The size of the receive socket buffer */
    RTI_INT32 max_receive_buffer_size;

    /* The maximum size of the message which can be received */
    RTI_INT32 max_message_size;

    /* The maximum TTL */
    RTI_INT32 multicast_ttl;

#ifndef RTI_CERT
    struct UDP_NatEntrySeq nat;
#endif

    /* The interface table if interfaces are added manually */
    struct UDP_InterfaceTableEntrySeq if_table;

    /* The network interface to use to send to multicast */
    REDA_String_T multicast_interface;

    /* If this should be considered the default UDP interfaces if
     * no other UDP interface is found to handle a route
     */
    RTI_BOOL is_default_interface;

    /* Disable reading of available network interfaces using system
     * information and instead rely on the manually configured
     * interface table
     */
    RTI_BOOL disable_auto_interface_config;

    /* Thread properties for each receive thread created by this
     * NETIO interface.
     */
    struct OSAPI_ThreadProperty recv_thread;
```

(continues on next page)

(continued from previous page)

```
    /* Bind to specific interfaces
         */
    RTI_BOOL enable_interface_bind;

    struct UDP_TransformRuleSeq source_rules;

    /* Rules for how to transform sent UDP payloads based on the
     * destination address.
     */
    struct UDP_TransformRuleSeq destination_rules;

    /* Determines how regular UDP is supported when transformations
     * are supported.
     */
    UDP_TransformUdpMode_T transform_udp_mode;

    /* The locator to use for locators that have transformations.
     */
    RTI_INT32 transform_locator_kind;
};
```

allow_interface

The allow_interface string sequence determines which interfaces are allowed to be used for communication. Each string element is the name of a network interface, such as "en0" or "eth1".

If this sequence is empty, all interface names pass the allow test. The default value is empty. Thus, all interfaces are allowed.

deny_interface

The deny_interface string sequence determines which interfaces are not allowed to be used for communication. Each string element is the name of a network interface, such as "en0" or "eth1".

If this sequence is empty, the test is false. That is, the interface is allowed. Note that the deny list is checked *after* the allow list. Thus, if an interface appears in both, it is denied. The default value is empty, thus no interfaces are denied.

max_send_buffer_size

The max_send_buffer_size is the maximum size of the send socket buffer and it *must* be at least as big as the largest sample. Typically, this buffer should be a multiple of the maximum number of samples that can be sent at any given time. The default value is 256KB.

max_receive_buffer_size

The max_receive_buffer_size is the maximum size of the receive socket buffer and it *must* be at least as big as the largest sample. Typically, this buffer should be a multiple of the maximum number of samples that can be received at any given time. The default value is 256KB.

max_message_size

The max_message_size is the maximum size of the message which can be received, including any packet overhead. The default value is 65507 bytes.

multicast_ttl

The multicast_ttl is the Multicast Time-To-Live (TTL). This value is only used for multicast. It limits the number of hops a packet can pass through before it is dropped by a router. The default value is 1.

nat

Connext DDS Micro supports firewalls with NAT. However, this feature has limited use and only supports translation between a private and public IP address. UDP ports are not translated. Furthermore, because *Connext DDS Micro* does not support any hole punching technique or WAN server, this feature is only useful when the private and public address mapping is static and known in advance. For example, to test between an Android emulator and the host, the following configuration can be used:

```
UDP_NatEntrySeq_set_maximum(&udp_property->nat,2);
UDP_NatEntrySeq_set_length(&udp_property->nat,2);

/* Translate the local emulator  eth0 address 10.10.2.f:7410 to
 * 127.0.0.1:7410. This ensures that the address advertised by the
 * emulator to the host machine is the host's loopback interface, not
 * the emulator's host interface
 */
UDP_NatEntrySeq_get_reference(&udp_property->nat,0)->
                            local_address.kind = NETIO_ADDRESS_KIND_UDPv4;
UDP_NatEntrySeq_get_reference(&udp_property->nat,0)->
                            local_address.port = 7410;
UDP_NatEntrySeq_get_reference(&udp_property->nat,0)->
                            local_address.value.ipv4.address = 0x0a00020f;

UDP_NatEntrySeq_get_reference(&udp_property->nat,0)->
                            public_address.kind = NETIO_ADDRESS_KIND_UDPv4;
UDP_NatEntrySeq_get_reference(&udp_property->nat,0)->
                            public_address.port = 7410;
UDP_NatEntrySeq_get_reference(&udp_property->nat,0)->
                            public_address.value.ipv4.address = 0x7f000001;

/* Translate the local emulator  eth0 address 10.10.2.f:7411 to
 * 127.0.0.1:7411. This ensures that the address advertised by the
 * emulator to the host machine is the host's loopback interface
 */
UDP_NatEntrySeq_get_reference(&udp_property->nat,1)->
                            local_address.kind = NETIO_ADDRESS_KIND_UDPv4;
UDP_NatEntrySeq_get_reference(&udp_property->nat,1)->
                            local_address.port = 7411;
UDP_NatEntrySeq_get_reference(&udp_property->nat,1)->
```

(continues on next page)

(continued from previous page)

```
                              local_address.value.ipv4.address = 0x0a00020f;

UDP_NatEntrySeq_get_reference(&udp_property->nat,1)->
                              public_address.kind = NETIO_ADDRESS_KIND_UDPv4;
UDP_NatEntrySeq_get_reference(&udp_property->nat,1)->
                              public_address.port = 7411;
UDP_NatEntrySeq_get_reference(&udp_property->nat,1)->
                              public_address.value.ipv4.address = 0x7f000001;
```

if_table

The if_table provides a method to manually configure which interfaces are available for use; for example, when using IP stacks that do not support reading interface lists. The following example shows how to manually configure the interfaces.

```
/* The arguments to the UDP_InterfaceTable_add_entry functions are:
 * The if_table itself
 * The network address of the interface
 * The netmask of the interface
 * The name of the interface
 * Interface flags. Valid flags are:
 *   UDP_INTERFACE_INTERFACE_UP_FLAG        - The interface is UP
 *   UDP_INTERFACE_INTERFACE_MULTICAST_FLAG - The interface supports multicast
 */
if (!UDP_InterfaceTable_add_entry(&udp_property->if_table,
                                  0x7f000001,0xff000000,"loopback",
                                  UDP_INTERFACE_INTERFACE_UP_FLAG |
                                  UDP_INTERFACE_INTERFACE_MULTICAST_FLAG))

{
    /* Error */
}
```

multicast_interface

The multicast_interface may be used to select a particular network interface to be used to send multicast packets. The default value is any interface (that is, the OS selects the interface).

is_default_interface

The is_default_interface flag is used to indicate that this *Connext DDS Micro* network transport shall be used if no other transport is found. The default value is **RTI_TRUE**.

disable_auto_interface_config

Normally, the UDP transport will try to read out the interface list (on platforms that support it). Setting disable_auto_interface_config to **RTI_TRUE** will prevent the UDP transport from reading the interface list.

recv_thread

The recv_thread field is used to configure all the receive threads. Please refer to *Threading Model* for details.

enable_interface_bind

When this is set to **TRUE** the UDP transport binds each receive port to a specific interface when the allow_interface/deny_interface lists are non-empty. This allows multiple UDP transports to be used by a single *DomainParticipant* at the expense of an increased number of threads. This property is ignored when transformations are enabled and the allow_interface/deny_interface lists are non-empty.

source_rules

Rules for how to transform received UDP payloads based on the source address.

destination_rules

Rules for how to transform sent UDP payloads based on the destination address.

transform_udp_mode

Determines how regular UDP is supported when transformations are supported. When transformations are enabled the default value is **UDP_TRANSFORM_UDP_MODE_DISABLED**.

transform_locator_kind

The locator to use for locators that have transformations. When transformation rules have been enabled, they are announced as a vendor specific locator. This property overrides this value.

NOTE: Changing this value may prevent communication.

UDP Transformations

The UDP transform feature enables custom transformation of incoming and outgoing UDP payloads based on transformation rules between a pair of source and destination IP addresses. Some examples of transformations are encrypted data or logging.

This section explains how to implement and use transformations in an application and is organized as follows:

- *Overview*
- *Creating a Transformation Library*
- *Creating Transformation Rules*
- *Interoperability*
- *Error Handling*
- *Example Code*

- *Examples*
- *OS Configuration*

Overview

The UDP transformation feature enables custom transformation of incoming and outgoing UDP payloads. For the purpose of this section, a UDP payload is defined as a sequence of octets sent or received as a single UDP datagram excluding UDP headers – typically UDP port numbers – and trailers, such as the optional used checksum.

An outgoing payload is the UDP payload passed to the network stack. The transformation feature allows a custom transformation of this payload just before it is sent. The UDP transport receives payloads to send from an upstream layer. In *Connext DDS Micro* this layer is typically RTPS, which creates payloads containing one or more RTPS messages. The transformation feature enables transformation of the entire RTPS payload before it is passed to the network stack.

The same RTPS payload may be sent to one or more locators. A locator identifies a destination address, such as an IPv4 address, a port, such as a UDP port, and a transport kind. The address and port are used by the UDP transport to reach a destination. However, only the destination address is used to determine which transformation to apply.

An incoming payload is the UDP payload received from the network stack. The transformation feature enables transformation of the UDP payload received from the network stack *before* it is passed to the upstream interface, typically RTPS. The UDP transport only receives payloads destined for one of its network interface addresses, but may receive UDP payloads destined for many different ports. The transformation does not take a port into account, only the source address. In *Connext DDS Micro* the payload is typically a RTPS payload containing one or more RTPS messages.

UDP transformations are registered with *Connext DDS Micro* and used by the UDP transport to determine how to transform payloads based on a source or destination address. Please refer to *Creating a Transformation Library* for details on how to implement transformations and *Creating Transformation Rules* for how to add rules.

Transformations are local resources. There is no exchange between different UDP transports regarding what a transformation does to a payload. This is considered a-priori knowledge and depends on the implementation of the transformation. Any negotiation of e.g. keys must be handled before the UDP transport is registered. Thus, if a sender and receiver do not apply consistent rules, they may not be able to communicate, or incorrect data may result. Note that while information is typically in the direction from a *DataWriter* to a *DataReader*, a reliable *DataReader* also send protocol data to a *DataWriter*. These messages are also transformed.

Network Interface Selection

When a *DomainParticipant* is created, it first creates an instance of each transport configured in the DomainParticipantQos::transports QoS policy. Thus, each UDP transport registered with *Connext DDS Micro* must have a unique name (up to 7 characters). Each registered transport can be configured to use all or some of the available interfaces using the allow_interface and deny_interface properties. The registered transports may now be used for either discovery data (specified in DomainParticipantQos::discovery), user_traffic (specified in DomainParticipantQos::user_traffic) or both. The *DomainParticipant* also queries the transport for which addresses it is capable of sending to.

When a participant creates multiple instances of the UDP transport, it is important that instances use non-overlapping networking interface resources.

Data Reception

Which transport to use for discovery data is determined by the DomainParticipantQos::discovery QoS policy. For each transport listed, the *DomainParticipant* reserves a network address to listen to. This network address is sent as part of the discovery data and is used by other *DomainParticipants* as the address to send discovery data for this *DomainParticipant*. Because a UDP transformation only looks at source and destination addresses, if different transformations are needed for discovery and user-data, different UDP transport registrations must be used and hence different network interfaces.

Data Transmission

Which address to send data to is based on the locators received as part of discovery and the peer list.

Received locators are analyzed and a transport locally registered with a *DomainParticipant* is selected based on the locator kind, address and mask. The first matching transport is selected. If a matching transport is not found, the locator is discarded.

NOTE: A transport is not a matching criteria at the same level as a QoS policy. If a discovered entity requests user data on a transport that doesn't exist, it is not unmatched.

The peer list, as specified by the application, is a list of locators to send participant discovery announcements to. If the transport to use is not specified, e.g. "udp1@192.168.1.1", but instead "192.168.1.1", then all transports that understand this address will send to it. Thus, in this case the latter is used, and two different UDP transports are registered; they will both send to the same address. However, one transport may send transformed data and the other may not depending on the destination address.

Creating a Transformation Library

The transformation library is responsible for creating and performing transformations. Note that a library is a logical concept and does not refer to an actual library in, for example, UNIX. A library in this context is a collection of routines that together creates, manages, and performs transformations. How these routines are compiled and linked with an application using *Connext DDS Micro* is out of scope of this section.

The transformation library must be registered with *Connext DDS Micro*'s run-time and must implement the required interfaces. This ensures proper life-cycle management of transformation resources as well as clear guidelines regarding concurrency and memory management.

From *Connext DDS Micro*'s run-time point of view, the transformation library must implement methods so that:

- A library can be initialized.

- A library can be instantiated.
- An instance of the library performs and manages transformations.

The first two tasks are handled by *Connext DDS Micro*'s run-time factory interface which is common for all libraries managed by *Connext DDS Micro*. The third task is handled by the transformation interface, which is specific to UDP transformations.

The following describes the relationship between the different interfaces:

- A library is initialized once when it is registered with *Connext DDS Micro*.
- A library is finalized once when it is unregistered from *Connext DDS Micro*.
- Multiple library instances can be created. If a library is used twice, for example registered with two different transports, two different library contexts are created using the factory interface. *Connext DDS Micro* assumes that concurrent access to two different instances is allowed.
- Different instances of the library can be deleted independently. An instance is deleted using the factory interface.
- A library instance creates specific source or destination transformations. Each transformation is expected to transform a payload to exactly one destination or from one source.

The following relationship is true between the UDP transport and a UDP transformation library:

- Each registered UDP transport may make use of one or more UDP transformation libraries.
- A DDS *DomainParticipant* creates one instance of each registered UDP transport.
- Each instance of the UDP transport creates one instance of each enabled transformation library registered with the UDP transport.
- Each Transformation rule created by the UDP transport creates one send or one receive transformation.

Creating Transformation Rules

Transformation rules decide how a payload should be transformed based on either a source or destination address. Before a UDP transport is registered, it must be configured with the transformation libraries to use, as well as which library to use for each source and destination address. For each UDP payload sent or received, an instance of the UDP transport searches for a matching source or destination rule to determine which transformation to apply.

The transformation rules are added to the UDP_InterfaceFactoryProperty before registration takes place.

If no transformation rules have been configured, all payloads are treated as regular UDP packets.

If no send rules have been asserted, the payload is sent as is. If all outgoing messages are to be transformed, a single entry is sufficient (address = 0, mask = 0).

If no receive rules have been asserted, it is passed upstream as is. If all incoming messages are to be transformed, a single entry is sufficient (address = 0, mask = 0).

If no matching rule is found, the packet is dropped and an error is logged.

NOTE: UDP_InterfaceFactoryProperty is immutable after the UDP transport has been registered.

Interoperability

When the UDP transformations has enabled at least one transformation, it will only inter-operate with another UDP transport which also has at least one transformation.

UDP transformations does not interoperate with *RTI Connext DDS Professional*.

Error Handling

The transformation rules are applied on a local basis and correctness is based on configuration. It is not possible to detect that a peer participant is configured for different behavior and errors cannot be detected by the UDP transport itself. However, the transformation interface can return errors which are logged.

Example Code

Example Header file MyUdpTransform.h:

```
#ifndef MyUdpTransform_h
#define MyUdpTransform_h

#include "rti_me_c.h"
#include "netio/netio_udp.h"
#include "netio/netio_interface.h"

struct MyUdpTransformFactoryProperty
{
    struct RT_ComponentFactoryProperty _parent;
};

extern struct RT_ComponentFactoryI*
MyUdpTransformFactory_get_interface(void);

extern RTI_BOOL
MyUdpTransformFactory_register(RT_Registry_T *registry,
                               const char *const name,
                               struct MyUdpTransformFactoryProperty *property);

extern RTI_BOOL
MyUdpTransformFactory_unregister(RT_Registry_T *registry,
                  const char *const name,
                  struct MyUdpTransformFactoryProperty **);

#endif
```

Example Source file MyUdpTransform.c:

```
/*ce
 * \file
```

(continues on next page)

(continued from previous page)

```
 * \defgroup UDPTransformExampleModule MyUdpTransform
 * \ingroup UserManuals_UDPTransform
 * \brief UDP Transform Example
 *
 * \details
 *
 * The UDP interface is implemented as a NETIO interface and NETIO interface
 * factory.
 */

 /*ce \addtogroup UDPTransformExampleModule
  * @{
  */
#include <stdio.h>

#include "MyUdpTransform.h"


/*ce
 * \brief The UDP Transformation factory class
 *
 * \details
 * All Transformation components must have a factory. A factory creates one
 * instance of the component as needed. In the case of UDP transformations,
 * \rtime creates one instance per UDP transport instance.
 */
struct MyUdpTransformFactory
{
    /*ce
     * \brief Base-class. All \rtime Factories must inherit from RT_ComponentFactory.
     */
    struct RT_ComponentFactory _parent;

    /*ce
     * \brief A pointer to the properties of the factory.
     *
     * \details
     *
     * When a factory is registered with \rtime it can be registered with
     * properties specific to the component. However \rtime does not
     * make a copy ( that would require additional methods). Furthermore, it
     * may not be desirable to make a copy. Instead, this decision is
     * left to the implementer of the component. \rtime does not access
     * any custom properties.
     */
    struct MyUdpTransformFactoryProperty *property;
};

/*ce
 * \brief The custom UDP transformation class.
 *
```

(continues on next page)

(continued from previous page)

```
 * \details
 * The MyUdpTransformFactory creates one instance of this class for each
 * UDP interface created. In this example one packet buffer (NETIO_Packet_T),
 * is allocated and a buffer to hold the transformed data (\ref buffer)
 *
 * Only one transformation can be done at a time and it is synchronous. Thus,
 * it is sufficient with one buffer to transform input and output per
 * instance of the MyUdpTransform.
 */
struct MyUdpTransform
{
    /*ce
     * \brief Base-class. All UDP transforms must inherit from UDP_Transform
     */
    struct UDP_Transform _parent;

    /*ce \brief A reference to its own factory, if properties must be accessed
     */
    struct MyUdpTransformFactory *factory;

    /*ce \brief NETIO_Packet to hold a transformed payload.
     *
     * \details
     *
     * \rtime uses a NETIO_Packet_T to abstract data payload and this is
     * what is being passed betweem the UDP transport and the transformation.
     * The transformation must convert a payload into a NETIO_Packet. This
     * is done with NETIO_Packet_initialize_from. This function saves all
     * state except the payload buffer.
     */
    NETIO_Packet_T packet;

    /*ce \brief The payload to assign to NETIO_Packet_T
     *
     * \details
     *
     * A transformation cannot do in-place transformations because the input
     * buffer may be sent multiple times (for example due to reliability).
     * A transformation instance can only transform one buffer at a time
     * (send or receive). The buffer must be large enough to hold a transformed
     * payload. When the the transformation is created it receives a
     * \ref UDP_TransformProperty. This property has the max send and
     * receive buffers for transport and can be used to sise the buffer.
     * Please refer to \ref UDP_InterfaceFactoryProperty::max_send_message_size
     * and \ref UDP_InterfaceFactoryProperty::max_message_size.
     */
    char *buffer;

    /*ce \brief The maximum length of the buffer. NOTE: The buffer must
     * be 1 byte larger than the largest buffer.
     */
```

(continues on next page)

(continued from previous page)

```
    RTI_SIZE_T max_buffer_length;
};

/*ce \brief Forward declaration of the interface implementation
 */
static struct UDP_TransformI MyUdpTransform_fv_Intf;

/*ce \brief Forward declaration of the interface factory implementation
 */
static struct RT_ComponentFactoryI MyUdpTransformFactory_fv_Intf;

/*ce \brief Method to create an instance of MyUdpTransform
 *
 * \param[in] factory The factory creating this instance
 * \param[in] property Generic UDP_Transform properties
 *
 * \return A pointer to MyUdpTransform on sucess, NULL on failure.
 */
RTI_PRIVATE struct MyUdpTransform*
MyUdpTransform_create(struct MyUdpTransformFactory *factory,
                      const struct UDP_TransformProperty *const property)
{
    struct MyUdpTransform *t;

    OSAPI_Heap_allocate_struct(&t, struct MyUdpTransform);
    if (t == NULL)
    {
        return NULL;
    }

    /* All component instances must initialize the parent using this
     * call.
     */
    RT_Component_initialize(&t->_parent._parent,
                            &MyUdpTransform_fv_Intf._parent,
                            0,
                            (property ? &property->_parent : NULL),
                            NULL);

    t->factory = factory;

    /* Allocate a buffer that is the larger of the send and receive
     * size.
     */
    t->max_buffer_length = property->max_receive_message_size;
    if (property->max_send_message_size > t->max_buffer_length )
    {
        t->max_buffer_length = property->max_send_message_size;
    }

    /* Allocate 1 extra byte */
```

(continues on next page)

(continued from previous page)

```
    OSAPI_Heap_allocate_buffer(&t->buffer,t->max_buffer_length+1,
                               OSAPI_ALIGNMENT_DEFAULT);

    if (t->buffer == NULL)
    {
        OSAPI_Heap_free_struct(t);
        t = NULL;
    }

    return t;
}

/*ce \brief Method to delete an instance of MyUdpTransform
 *
 * \param[in] t Transformation instance to delete
 */
RTI_PRIVATE void
MyUdpTransform_delete(struct MyUdpTransform *t)
{
    OSAPI_Heap_free_buffer(t->buffer);
    OSAPI_Heap_free_struct(t);
}

/*ce \brief Method to create a transformation for an destination address
 *
 * \details
 *
 * For each asserted destination rule a transform is created by the transformation
 * instance. This method determines how a UDP payload is transformed before
 * it is sent to an address that matches destination & netmask.
 *
 * \param[in]  udptf       UDP Transform instance that creates the transformation
 * \param[out] context     Pointer to a transformation context
 * \param[in]  destination Destination address for the transformation
   \param[in]  netmask     The netmask to apply to this destination.
 * \param[in]  user_data   The user_data the rule was asserted with
 * \param[in]  property    UDP transform specific properties
 * \param[out] ec          User defined error code
 *
 * \return RTI_TRUE on success, RTI_FALSE on failure
 */
RTI_PRIVATE RTI_BOOL
MyUdpTransform_create_destination_transform(
                             UDP_Transform_T *const udptf,
                             void **const context,
                             const struct NETIO_Address *const destination,
                             const struct NETIO_Netmask *const netmask,
                             void *user_data,
                             const struct UDP_TransformProperty *const property,
                             RTI_INT32 *const ec)
{
```

(continues on next page)

(continued from previous page)

```
    struct MyUdpTransform *self = (struct MyUdpTransform*)udptf;
    UNUSED_ARG(self);
    UNUSED_ARG(destination);
    UNUSED_ARG(user_data);
    UNUSED_ARG(property);
    UNUSED_ARG(ec);
    UNUSED_ARG(netmask);

    /* Save the user-data to determine which transform to apply later */
    *context = (void*)user_data;

    return RTI_TRUE;
}

/*ce \brief Method to delete a transformation for an destination address
 *
 *
 * \param[in]  udptf       UDP Transform instance that created the transformation
 * \param[out] context     Pointer to a transformation context
 * \param[in]  destination Destination address for the transformation
 * \param[in]  netmask     The netmask to apply to this destination.
 * \param[out] ec          User defined error code
 *
 * \return RTI_TRUE on success, RTI_FALSE on failure
 */
RTI_PRIVATE RTI_BOOL
MyUdpTransform_delete_destination_transform(UDP_Transform_T *const udptf,
                                void *context,
                                const struct NETIO_Address *const destination,
                                const struct NETIO_Netmask *const netmask,
                                RTI_INT32 *const ec)
{
    UNUSED_ARG(udptf);
    UNUSED_ARG(context);
    UNUSED_ARG(destination);
    UNUSED_ARG(ec);
    UNUSED_ARG(netmask);

    return RTI_TRUE;
}

/*ce \brief Method to create a transformation for an source address
 *
 * \details
 *
 * For each asserted source rule a transform is created by the transformation
 * instance. This method determines how a UDP payload is transformed when
 * it is received from an address that matches source & netmask.
 *
 * \param[in]  udptf       UDP Transform instance that creates the transformation
 * \param[out] context     Pointer to a transformation context
```

(continues on next page)

(continued from previous page)

```
 * \param[in]  source      Destination address for the transformation
   \param[in]  netmask     The netmask to apply to this destination.
 * \param[in]  user_data   The user_data the rule was asserted with
 * \param[in]  property    UDP transform specific properties
 * \param[out] ec          User defined error code
 *
 * \return RTI_TRUE on success, RTI_FALSE on failure
 */
RTI_PRIVATE RTI_BOOL
MyUdpTransform_create_source_transform(UDP_Transform_T *const udptf,
                           void **const context,
                           const struct NETIO_Address *const source,
                           const struct NETIO_Netmask *const netmask,
                           void *user_data,
                           const struct UDP_TransformProperty *const property,
                           RTI_INT32 *const ec)
{
    struct MyUdpTransform *self = (struct MyUdpTransform*)udptf;
    UNUSED_ARG(self);
    UNUSED_ARG(source);
    UNUSED_ARG(user_data);
    UNUSED_ARG(property);
    UNUSED_ARG(ec);
    UNUSED_ARG(netmask);

    *context = (void*)user_data;

    return RTI_TRUE;
}

/*ce \brief Method to delete a transformation for an source address
 *
 *
 * \param[in]  udptf       UDP Transform instance that created the transformation
 * \param[out] context     Pointer to a transformation context
 * \param[in]  source      Source address for the transformation
 * \param[in]  netmask     The netmask to apply to this destination.
 * \param[out] ec          User defined error code
 *
 * \return RTI_TRUE on success, RTI_FALSE on failure
 */
RTI_PRIVATE RTI_BOOL
MyUdpTransform_delete_source_transform(UDP_Transform_T *const udptf,
                                       void *context,
                                       const struct NETIO_Address *const source,
                                       const struct NETIO_Netmask *const netmask,
                                       RTI_INT32 *const ec)
{
    UNUSED_ARG(udptf);
    UNUSED_ARG(context);
    UNUSED_ARG(source);
```

(continues on next page)

(continued from previous page)

```
    UNUSED_ARG(ec);
    UNUSED_ARG(netmask);

    return RTI_TRUE;
}

/*ce \brief Method to transform data based on a source address
 *
 * \param[in]  udptf       UDP_Transform_T that performs the transformation
 * \param[in]  context     Reference to context created by \ref MyUdpTransform_create_
↪source_transform
 * \param[in]  source      Source address for the transformation
 * \param[in]  in_packet   The NETIO packet to transform
 * \param[out] out_packet  The transformed NETIO packet
 * \param[out] ec          User defined error code
 *
 * \return RTI_TRUE on success, RTI_FALSE on failure
 */
RTI_PRIVATE RTI_BOOL
MyUdpTransform_transform_source(UDP_Transform_T *const udptf,
                                void *context,
                                const struct NETIO_Address *const source,
                                const NETIO_Packet_T *const in_packet,
                                NETIO_Packet_T **out_packet,
                                RTI_INT32 *const ec)
{
    struct MyUdpTransform *self = (struct MyUdpTransform*)udptf;
    char *buf_ptr,*buf_end;
    char *from_buf_ptr,*from_buf_end;
    UNUSED_ARG(context);
    UNUSED_ARG(source);

    *ec = 0;

    /* Assigned the transform buffer to the outgoing packet
     * saving state from the incoming packet. In this case the
     * outgoing length is the same as the incoming. How to buffer
     * is filled in is of no interest to \rtime. All it cares about is
     * where it starts and where it ends.
     */
    if (!NETIO_Packet_initialize_from(
                                &self->packet,in_packet,
                                self->buffer,self->max_buffer_length,
                                0,NETIO_Packet_get_payload_length(in_packet)))
    {
        return RTI_FALSE;
    }

    *out_packet = &self->packet;

    buf_ptr = NETIO_Packet_get_head(&self->packet);
```

(continues on next page)

(continued from previous page)

```
    buf_end = NETIO_Packet_get_tail(&self->packet);
    from_buf_ptr = NETIO_Packet_get_head(in_packet);
    from_buf_end = NETIO_Packet_get_tail(in_packet);

    /* Perform a transformation based on the user-data */
    while (from_buf_ptr < from_buf_end)
    {
        if (context == (void*)1)
        {
            *buf_ptr = ~(*from_buf_ptr);
        }
        else if (context == (void*)2)
        {
            *buf_ptr = (*from_buf_ptr)+1;
        }

        ++buf_ptr;
        ++from_buf_ptr;
    }

    return RTI_TRUE;
}

/*ce \brief Method to transform data based on a destination address
 *
 * \param[in]  udptf       UDP_Transform_T that performs the transformation
 * \param[in]  context     Reference to context created by \ref MyUdpTransform_create_
↪destination_transform
 * \param[in]  destination Source address for the transformation
 * \param[in]  in_packet   The NETIO packet to transform
 * \param[out] packet_out  The transformed NETIO packet
 * \param[out] ec          User defined error code
 *
 * \return RTI_TRUE on success, RTI_FALSE on failure
 */
RTI_PRIVATE RTI_BOOL
MyUdpTransform_transform_destination(UDP_Transform_T *const udptf,
                                     void *context,
                                     const struct NETIO_Address *const destination,
                                     const NETIO_Packet_T *const in_packet,
                                     NETIO_Packet_T **packet_out,
                                     RTI_INT32 *const ec)
{
    struct MyUdpTransform *self = (struct MyUdpTransform*)udptf;
    char *buf_ptr,*buf_end;
    char *from_buf_ptr,*from_buf_end;
    UNUSED_ARG(context);
    UNUSED_ARG(destination);

    *ec = 0;
```

(continues on next page)

(continued from previous page)

```
    if (!NETIO_Packet_initialize_from(
                                &self->packet,in_packet,
                                self->buffer,8192,
                                0,NETIO_Packet_get_payload_length(in_packet)))
    {
        return RTI_FALSE;
    }

    *out_packet = &self->packet;

    buf_ptr = NETIO_Packet_get_head(&self->packet);
    buf_end = NETIO_Packet_get_tail(&self->packet);
    from_buf_ptr = NETIO_Packet_get_head(in_packet);
    from_buf_end = NETIO_Packet_get_tail(in_packet);

    while (from_buf_ptr < from_buf_end)
    {
        if (context == (void*)1)
        {
            *buf_ptr = ~(*from_buf_ptr);
        }
        else if (context == (void*)2)
        {
            *buf_ptr = (*from_buf_ptr)-1;
        }

        ++buf_ptr;
        ++from_buf_ptr;
    }

    return RTI_TRUE;
}

/*ce \brief Definition of the transformation interface
 */
RTI_PRIVATE struct UDP_TransformI MyUdpTransform_fv_Intf =
{
    RT_COMPONENTI_BASE,
    MyUdpTransform_create_destination_transform,
    MyUdpTransform_create_source_transform,
    MyUdpTransform_transform_source,
    MyUdpTransform_transform_destination,
    MyUdpTransform_delete_destination_transform,
    MyUdpTransform_delete_source_transform
};

/*ce \brief Method called by \rtime to create an instance of transformation
 */
MUST_CHECK_RETURN RTI_PRIVATE RT_Component_T*
MyUdpTransformFactory_create_component(struct RT_ComponentFactory *factory,
                        struct RT_ComponentProperty *property,
```

(continues on next page)

(continued from previous page)

```
                        struct RT_ComponentListener *listener)
{
    struct MyUdpTransform *t;
    UNUSED_ARG(listener);

    t = MyUdpTransform_create(
                (struct MyUdpTransformFactory*)factory,
                (struct UDP_TransformProperty*)property);

    return &t->_parent._parent;
}

/*ce \brief Method called by \rtime to delete an instance of transformation
 */
RTI_PRIVATE void
MyUdpTransformFactory_delete_component(
                                        struct RT_ComponentFactory *factory,
                                        RT_Component_T *component)
{
    UNUSED_ARG(factory);

    MyUdpTransform_delete((struct MyUdpTransform*)component);
}

/*ce \brief Method called by \rtime when a factory is registered
 */
MUST_CHECK_RETURN RTI_PRIVATE struct RT_ComponentFactory*
MyUdpTransformFactory_initialize(struct RT_ComponentFactoryProperty* property,
                                  struct RT_ComponentFactoryListener *listener)
{
    struct MyUdpTransformFactory *fac;
    UNUSED_ARG(property);
    UNUSED_ARG(listener);

    OSAPI_Heap_allocate_struct(&fac,struct MyUdpTransformFactory);

    fac->_parent._factory = &fac->_parent;
    fac->_parent.intf = &MyUdpTransformFactory_fv_Intf;
    fac->property = (struct MyUdpTransformFactoryProperty*)property;

    return &fac->_parent;
}

/*ce \brief Method called by \rtime when a factory is unregistered
 */
RTI_PRIVATE void
MyUdpTransformFactory_finalize(struct RT_ComponentFactory *factory,
                        struct RT_ComponentFactoryProperty **property,
                        struct RT_ComponentFactoryListener **listener)
{
    struct MyUdpTransformFactory *fac =
```

(continues on next page)

(continued from previous page)

```
            (struct MyUdpTransformFactory*)factory;

    UNUSED_ARG(property);
    UNUSED_ARG(listener);

    if (listener != NULL)
    {
        *listener = NULL;
    }

    if (property != NULL)
    {
        *property = (struct RT_ComponentFactoryProperty*)fac->property;
    }

    OSAPI_Heap_free_struct(factory);

    return;
}

/*ce \brief Definition of the factory interface
 */
RTI_PRIVATE struct RT_ComponentFactoryI MyUdpTransformFactory_fv_Intf =
{
    UDP_INTERFACE_INTERFACE_ID,
    MyUdpTransformFactory_initialize,
    MyUdpTransformFactory_finalize,
    MyUdpTransformFactory_create_component,
    MyUdpTransformFactory_delete_component,
    NULL
};

struct RT_ComponentFactoryI*
MyUdpTransformFactory_get_interface(void)
{
    return &MyUdpTransformFactory_fv_Intf;
}

/*ce \brief Method to register this transformation in a registry
 */
RTI_BOOL
MyUdpTransformFactory_register(RT_Registry_T *registry,
                            const char *const name,
                            struct MyUdpTransformFactoryProperty *property)
{
    return RT_Registry_register(registry, name,
                        MyUdpTransformFactory_get_interface(),
                        &property->_parent, NULL);
}

/*ce \brief Method to unregister this transformation from a registry
```

(continues on next page)

(continued from previous page)

```
 */
RTI_BOOL
MyUdpTransformFactory_unregister(RT_Registry_T *registry,
            const char *const name,
            struct MyUdpTransformFactoryProperty **property)
{
    return RT_Registry_unregister(registry, name,
                              (struct RT_ComponentFactoryProperty**)property,
                              NULL);
}

/*! @} */
```

Example configuration of rules:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "common.h"

void
MyAppApplication_help(char *appname)
{
    printf("%s [options]\n", appname);
    printf("options:\n");
    printf("-h                 - This text\n");
    printf("-domain <id>       - DomainId (default: 0)\n");
    printf("-udp_intf <intf>   - udp interface (no default)\n");
    printf("-peer <address>    - peer address (no default)\n");
    printf("-count <count>     - count (default -1)\n");
    printf("-sleep <ms>        - sleep between sends (default 1s)\n");
    printf("\n");
}

struct MyAppApplication*
MyAppApplication_create(const char *local_participant_name,
                     const char *remote_participant_name,
                     DDS_Long domain_id, char *udp_intf, char *peer,
                     DDS_Long sleep_time, DDS_Long count)
{
    DDS_ReturnCode_t retcode;
    DDS_DomainParticipantFactory *factory = NULL;
    struct DDS_DomainParticipantFactoryQos dpf_qos =
        DDS_DomainParticipantFactoryQos_INITIALIZER;
    struct DDS_DomainParticipantQos dp_qos =
        DDS_DomainParticipantQos_INITIALIZER;
    DDS_Boolean success = DDS_BOOLEAN_FALSE;
    struct MyAppApplication *application = NULL;
    RT_Registry_T *registry = NULL;
```

(continues on next page)

(continued from previous page)

```
    struct UDP_InterfaceFactoryProperty *udp_property = NULL;
    struct DPDE_DiscoveryPluginProperty discovery_plugin_properties =
        DPDE_DiscoveryPluginProperty_INITIALIZER;
    UNUSED_ARG(local_participant_name);
    UNUSED_ARG(remote_participant_name);

    /* Uncomment to increase verbosity level:
       OSAPILog_set_verbosity(OSAPI_LOG_VERBOSITY_WARNING);
     */
    application = (struct MyAppApplication *)malloc(sizeof(struct MyAppApplication));

    if (application == NULL)
    {
        printf("failed to allocate application\n");
        goto done;
    }

    application->sleep_time = sleep_time;
    application->count = count;

    factory = DDS_DomainParticipantFactory_get_instance();

    if (DDS_DomainParticipantFactory_get_qos(factory,&dpf_qos) != DDS_RETCODE_OK)
    {
        printf("failed to get number of components\n");
        goto done;
    }

    dpf_qos.resource_limits.max_components = 128;

    if (DDS_DomainParticipantFactory_set_qos(factory,&dpf_qos) != DDS_RETCODE_OK)
    {
        printf("failed to increase number of components\n");
        goto done;
    }

    registry = DDS_DomainParticipantFactory_get_registry(
                            DDS_DomainParticipantFactory_get_instance());

    if (!RT_Registry_register(registry, DDSHST_WRITER_DEFAULT_HISTORY_NAME,
                          WHSM_HistoryFactory_get_interface(), NULL, NULL))
    {
        printf("failed to register wh\n");
        goto done;
    }

    if (!RT_Registry_register(registry, DDSHST_READER_DEFAULT_HISTORY_NAME,
                          RHSM_HistoryFactory_get_interface(), NULL, NULL))
    {
        printf("failed to register rh\n");
        goto done;
```

(continues on next page)

(continued from previous page)

```
    }

    if (!MyUdpTransformFactory_register(registry,"T0",NULL))
    {
        printf("failed to register T0\n");
        goto done;
    }

    if (!MyUdpTransformFactory_register(registry,"T1",NULL))
    {
        printf("failed to register T0\n");
        goto done;
    }

    /* Configure UDP transport's allowed interfaces */
    if (!RT_Registry_unregister(registry, NETIO_DEFAULT_UDP_NAME, NULL, NULL))
    {
        printf("failed to unregister udp\n");
        goto done;
    }

    udp_property = (struct UDP_InterfaceFactoryProperty *)
                        malloc(sizeof(struct UDP_InterfaceFactoryProperty));
    if (udp_property == NULL)
    {
        printf("failed to allocate udp properties\n");
        goto done;
    }
    *udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    /* For additional allowed interface(s), increase maximum and length, and
       set interface below:
    */
    udp_property->max_send_message_size = 16384;
    udp_property->max_message_size = 32768;

    if (udp_intf != NULL)
    {
        REDA_StringSeq_set_maximum(&udp_property->allow_interface,1);
        REDA_StringSeq_set_length(&udp_property->allow_interface,1);
        *REDA_StringSeq_get_reference(&udp_property->allow_interface,0) =
                DDS_String_dup(udp_intf);
    }


    /* A rule that says: For payloads received from 192.168.10.* (netmask is
     * 0xffffff00), apply transformation T0.
     */
    if (!UDP_TransformRules_assert_source_rule(
            &udp_property->source_rules,
            0xc0a80ae8,0xffffff00,"T0",(void*)2))
```

(continues on next page)

(continued from previous page)

```
    {
        printf("Failed to assert source rule\n");
        goto done;
    }

    /* A rule that says: For payloads sent to 192.168.10.* (netmask is
     * 0xffffff00), apply transformation T0.
     */
    if (!UDP_TransformRules_assert_destination_rule(
            &udp_property->destination_rules,
            0xc0a80ae8,0xffffff00,"T0",(void*)2))
    {
        printf("Failed to assert source rule\n");
        goto done;
    }

    /* A rule that says: For payloads received from 192.168.20.* (netmask is
     * 0xffffff00), apply transformation T1.
     */
    if (!UDP_TransformRules_assert_source_rule(
            &udp_property->source_rules,
            0xc0a81465,0xffffff00,"T1",(void*)1))
    {
        printf("Failed to assert source rule\n");
        goto done;
    }

    /* A rule that says: For payloads received from 192.168.20.* (netmask is
     * 0xffffff00), apply transformation T1.
     */
    if (!UDP_TransformRules_assert_destination_rule(
            &udp_property->destination_rules,
            0xc0a81465,0xffffff00,"T1",(void*)1))
    {
        printf("Failed to assert source rule\n");
        goto done;
    }

    if (!RT_Registry_register(registry, NETIO_DEFAULT_UDP_NAME,
                        UDP_InterfaceFactory_get_interface(),
                        (struct RT_ComponentFactoryProperty*)udp_property, NULL))
    {
       printf("failed to register udp\n");
       goto done;
    }

    DDS_DomainParticipantFactory_get_qos(factory, &dpf_qos);
    dpf_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
    DDS_DomainParticipantFactory_set_qos(factory, &dpf_qos);

    if (peer == NULL)
```

(continues on next page)

(continued from previous page)

```
    {
        peer = "127.0.0.1"; /* default to loopback */
    }

    if (!RT_Registry_register(registry,
                              "dpde",
                              DPDE_DiscoveryFactory_get_interface(),
                              &discovery_plugin_properties._parent,
                              NULL))
    {
        printf("failed to register dpde\n");
        goto done;
    }

    if (!RT_ComponentFactoryId_set_name(&dp_qos.discovery.discovery.name,"dpde"))
    {
        printf("failed to set discovery plugin name\n");
        goto done;
    }

    DDS_StringSeq_set_maximum(&dp_qos.discovery.initial_peers,1);
    DDS_StringSeq_set_length(&dp_qos.discovery.initial_peers,1);
    *DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers,0) = DDS_String_
↪dup(peer);

    DDS_StringSeq_set_maximum(&dp_qos.discovery.enabled_transports,1);
    DDS_StringSeq_set_length(&dp_qos.discovery.enabled_transports,1);

    /* Use network interface 192.168.10.232 for discovery. TO is used for
     * discovery
     */
    *DDS_StringSeq_get_reference(&dp_qos.discovery.enabled_transports,0) = DDS_String_
↪dup("_udp://192.168.10.232");

    DDS_StringSeq_set_maximum(&dp_qos.user_traffic.enabled_transports,1);
    DDS_StringSeq_set_length(&dp_qos.user_traffic.enabled_transports,1);

    /* Use network interface 192.168.20.101 for user-data. T1 is used for
     * this interface.
     */
    *DDS_StringSeq_get_reference(&dp_qos.user_traffic.enabled_transports,0) = DDS_String_
↪dup("_udp://192.168.20.101");

    /* if there are more remote or local endpoints, you need to increase these limits */
    dp_qos.resource_limits.max_destination_ports = 32;
    dp_qos.resource_limits.max_receive_ports = 32;
    dp_qos.resource_limits.local_topic_allocation = 1;
    dp_qos.resource_limits.local_type_allocation = 1;
    dp_qos.resource_limits.local_reader_allocation = 1;
    dp_qos.resource_limits.local_writer_allocation = 1;
    dp_qos.resource_limits.remote_participant_allocation = 8;
```

(continues on next page)

(continued from previous page)

```
    dp_qos.resource_limits.remote_reader_allocation = 8;
    dp_qos.resource_limits.remote_writer_allocation = 8;

    application->participant =
        DDS_DomainParticipantFactory_create_participant(factory, domain_id,
                                                        &dp_qos, NULL,
                                                        DDS_STATUS_MASK_NONE);

    if (application->participant == NULL)
    {
        printf("failed to create participant\n");
        goto done;
    }

    sprintf(application->type_name, "HelloWorld");
    retcode = DDS_DomainParticipant_register_type(application->participant,
                                          application->type_name,
                                          HelloWorldTypePlugin_get());
    if (retcode != DDS_RETCODE_OK)
    {
        printf("failed to register type: %s\n", "test_type");
        goto done;
    }

    sprintf(application->topic_name, "HelloWorld");
    application->topic =
        DDS_DomainParticipant_create_topic(application->participant,
                                           application->topic_name,
                                           application->type_name,
                                           &DDS_TOPIC_QOS_DEFAULT, NULL,
                                           DDS_STATUS_MASK_NONE);

    if (application->topic == NULL)
    {
        printf("topic == NULL\n");
        goto done;
    }

    success = DDS_BOOLEAN_TRUE;

  done:

    if (!success)
    {
        if (udp_property != NULL)
        {
            free(udp_property);
        }
        free(application);
        application = NULL;
    }
```

(continues on next page)

(continued from previous page)

```
    return application;
}

DDS_ReturnCode_t
MyAppApplication_enable(struct MyAppApplication * application)
{
    DDS_Entity *entity;
    DDS_ReturnCode_t retcode;

    entity = DDS_DomainParticipant_as_entity(application->participant);

    retcode = DDS_Entity_enable(entity);
    if (retcode != DDS_RETCODE_OK)
    {
        printf("failed to enable entity\n");
    }

    return retcode;
}

void
MyAppApplication_delete(struct MyAppApplication *application)
{
    DDS_ReturnCode_t retcode;
    RT_Registry_T *registry = NULL;

    retcode = DDS_DomainParticipant_delete_contained_entities(application->participant);
    if (retcode != DDS_RETCODE_OK)
    {
        printf("failed to delete conteined entities (retcode=%d)\n",retcode);
    }

    if (DDS_DomainParticipant_unregister_type(application->participant,
                    application->type_name) != HelloWorldTypePlugin_get())
    {
        printf("failed to unregister type: %s\n", application->type_name);
        return;
    }

    retcode = DDS_DomainParticipantFactory_delete_participant(
                                DDS_DomainParticipantFactory_get_instance(),
                                application->participant);

    if (retcode != DDS_RETCODE_OK)
    {
        printf("failed to delete participant: %d\n", retcode);
        return;
    }

    registry = DDS_DomainParticipantFactory_get_registry(
```

(continues on next page)

(continued from previous page)

```
                                    DDS_DomainParticipantFactory_get_instance());

    if (!RT_Registry_unregister(registry, "dpde", NULL, NULL))
    {
        printf("failed to unregister dpde\n");
        return;
    }
    if (!RT_Registry_unregister(registry, DDSHST_READER_DEFAULT_HISTORY_NAME, NULL,␣
↪NULL))
    {
        printf("failed to unregister rh\n");
        return;
    }
    if (!RT_Registry_unregister(registry, DDSHST_WRITER_DEFAULT_HISTORY_NAME, NULL,␣
↪NULL))
    {
        printf("failed to unregister wh\n");
        return;
    }

    free(application);

    DDS_DomainParticipantFactory_finalize_instance();
}
```

Examples

The following examples illustrate how this feature can be used in a system with a mixture of different types of UDP transport configurations.

For the purpose of the examples, the following terminology is used:

- Plain communication – No transformations have been applied.
- Transformed User Data – Only the user-data is transformed, discovery is plain.
- Transformed Discovery – Only the discovery data is transformed, user-data is plain.
- Transformed Data – Both discovery and user-data are transformed. Unless stated otherwise the transformations are different.

A transformation Tn is a transformation such that an outgoing payload transformed with Tn can be transformed back to its original state by applying Tn to the incoming data.

A network interface can be either physical or virtual.

Plain Communication Between 2 Nodes

In this system two Nodes, A and B, are communicating with plain communication. Node A has one interface, a0, and Node B has one interface, b0.

Node A:

- Register the UDP transport Ua with allow_interface = a0.
- DomainParticipantQos.transports.enabled_transports = “Ua”
- DomainParticipantQos.discovery.enabled_transports = ”Ua://”
- DomainParticipantQos.user_data.enabled_transports = ”Ua://”

Node B:

- Register the UDP transport Ub with allow_interface = b0.
- DomainParticipantQos.transports.enabled_transports = “Ub”
- DomainParticipantQos.discovery.enabled_transports = ”Ub://”
- DomainParticipantQos.user_data.enabled_transports = ”Ub://”

Transformed User Data Between 2 Nodes

In this system two Nodes, A and B, are communicating with transformed user data. Node A has two interfaces, a0 and a1, and Node B has two interfaces, b0 and b1. Since each node has only one peer, a single transformation is sufficient.

Node A:

- Add a destination transformation T0 to Ua0, indicating that all sent data is transformed with T0.
- Add a source transformation T1 to Ua0, indicating that all received data is transformed with T1.
- Register the UDP transport Ua0 with allow_interface = a0.
- Register the UDP transport Ua1 with allow_interface = a1.
- No transformations are registered with Ua1.
- DomainParticipantQos.transports.enabled_transports = “Ua0”,”Ua1”
- DomainParticipantQos.discovery.enabled_transports = ”Ua1://”
- DomainParticipantQos.user_traffic.enabled_transports = ”Ua0://”

Node B:

- Add a destination transformation T1 to Ub0, indicating that all sent data is transformed with T1.
- Add a source transformation T0 to Ub0, indicating that all received data is transformed with T0.
- Register the UDP transport Ub0 with allow_interface = b0.
- Register the UDP transport Ub1 with allow_interface = b1.
- No transformations are registered with Ub1.
- DomainParticipantQos.transports.enabled_transports = “Ub0”,”Ub1”

- DomainParticipantQos.discovery.enabled_transports = ”Ub1://”
- DomainParticipantQos.user_traffic.enabled_transports = ”Ub0://”

Ua0 and Ub0 perform transformations and are used for user-data. Ua1 and Ub1 are used for discovery and no transformations takes place.

Transformed Discovery Data Between 2 Nodes

In this system two Nodes, A and B, are communicating with transformed user data. Node A has two interfaces, a0 and a1, and Node B has two interfaces, b0 and b1. Since each node has only one peer, a single transformation is sufficient.

Node A:

- Add a destination transformation T0 to Ua0, indicating that all sent data is transformed with T0.
- Add a source transformation T1 to Ua0, indicating that all received data is transformed with T1.
- Register the UDP transport Ua0 with allow_interface = a0.
- Register the UDP transport Ua1 with allow_interface = a1.
- No transformations are registered with Ua1.
- DomainParticipantQos.transports.enabled_transports = “Ua0”,”Ua1”
- DomainParticipantQos.discovery.enabled_transports = ”Ua0://”
- DomainParticipantQos.user_data.enabled_transports = ”Ua1://”

Node B:

- Add a destination transformation T1 to Ub0, indicating that all sent data is transformed with T1.
- Add a source transformation T0 to Ub0, indicating that all received data is transformed with T0.
- Register the UDP transport Ub0 with allow_interface = b0.
- Register the UDP transport Ub1 with allow_interface = b1.
- No transformations are registered with Ub1.
- DomainParticipantQos.transports.enabled_transports = “Ub0”,”Ub1”
- DomainParticipantQos.discovery.enabled_transports = ”Ub0://”
- DomainParticipantQos.user_data.enabled_transports = ”Ub1://”

Ua0 and Ub0 perform transformations and are used for discovery. Ua1 and Ub1 are used for user-data and no transformation takes place.

Transformed Data Between 2 Nodes (same transformation)

In this system two Nodes, A and B, are communicating with transformed data using the same transformation for user and discovery data. Node A has one interface, a0, and Node B has one interface, b0.

Node A:

- Add a destination transformation T0 to Ua0, indicating that all sent data is transformed with T0.
- Add a source transformation T1 to Ua0, indicating that all received data is transformed with T1.
- Register the UDP transport Ua0 with allow_interface = a0.
- DomainParticipantQos.transports.enabled_transports = “Ua0”
- DomainParticipantQos.discovery.enabled_transports = ”Ua0://”
- DomainParticipantQos.user_data.enabled_transports = ”Ua0://”

Node B:

- Add a destination transformation T1 to Ub0, indicating that all sent data is transformed with T1.
- Add a source transformation T0 to Ub0, indicating that all received data is transformed with T0.
- Register the UDP transport Ub0 with allow_interface = b0.
- DomainParticipantQos.transports.enabled_transports = “Ub0”
- DomainParticipantQos.discovery.enabled_transports = ”Ub0://”
- DomainParticipantQos.user_data.enabled_transports = ”Ub0://”

Ua0 and Ub0 performs transformations and are used for discovery and for user-data.

Transformed Data Between 2 Nodes (different transformations)

In this system two Nodes, A and B, are communicating with transformed data using different transformations for user and discovery data. Node A has two interfaces, a0 and a1, and Node B has two interfaces, b0 and b1.

Node A:

- Add a destination transformation T0 to Ua0, indicating that all sent data is transformed with T0.
- Add a source transformation T1 to Ua0, indicating that all received data is transformed with T1.
- Add a destination transformation T2 to Ua1, indicating that all sent data is transformed with T2.

- Add a source transformation T3 to Ua1, indicating that all received data is transformed with T3.
- Register the UDP transport Ua0 with allow_interface = a0.
- Register the UDP transport Ua1 with allow_interface = a1.
- DomainParticipantQos.transports.enabled_transports = "Ua0","Ua1"
- DomainParticipantQos.discovery.enabled_transports = "Ua0://"
- DomainParticipantQos.user_data.enabled_transports = "Ua1://"

Node B:

- Add a destination transformation T1 to Ub0, indicating that all sent data is transformed with T1.
- Add a source transformation T0 to Ub0, indicating that all received data is transformed with T0.
- Add a destination transformation T3 to Ub1, indicating that all sent data is transformed with T3.
- Add a source transformation T2 to Ub1, indicating that all received data is transformed with T2.
- Register the UDP transport Ub0 with allow_interface = b0.
- Register the UDP transport Ub1 with allow_interface = b1.
- DomainParticipantQos.transports.enabled_transports = "Ub0","Ub1"
- DomainParticipantQos.discovery.enabled_transports = "Ub0://"
- DomainParticipantQos.user_data.enabled_transports = "Ub1://"

Ua0 and Ub0 perform transformations and are used for discovery. Ua1 and Ub1 perform transformations and are used for user-data.

OS Configuration

In systems with several network interfaces, *Connext DDS Micro* cannot ensure which network interface should be used to send a packet. Depending on the UDP transformations configured, this might be a problem.

To illustrate this problem, let's assume a system with two nodes, A and B. Node A has two network interfaces, a0 and a1, and Node B has two network interfaces, b0 and b1. In this system, Node A is communicating with Node B using a transformation for discovery and a different transformation for user data.

Node A:

- Add a destination transformation T0 to Ua0, indicating that sent data to b0 is transformed with T0.
- Add a source transformation T1 to Ua0, indicating that received data from b0 is transformed with T1.

- Add a destination transformation T2 to Ua1, indicating that sent data to b1 is transformed with T2.
- Add a source transformation T3 to Ua1, indicating that received data from b1 is transformed with T3.
- Register the UDP transport Ua0 with allow_interface = a0.
- Register the UDP transport Ua1 with allow_interface = a1.
- DomainParticipantQos.transports.enabled_transports = “Ua0”,”Ua1”
- DomainParticipantQos.discovery.enabled_transports = ”Ua0://”
- DomainParticipantQos.user_data.enabled_transports = ”Ua1://”

Node B:

- Add a destination transformation T1 to Ub0, indicating that sent data to a0 is transformed with T1.
- Add a source transformation T0 to Ub0, indicating that received data from a0 transformed with T0.
- Add a destination transformation T3 to Ub1, indicating that sent data to a1 is transformed with T3.
- Add a source transformation T2 to Ub1, indicating that received data from a1 transformed with T2.
- Register the UDP transport Ub0 with allow_interface = b0.
- Register the UDP transport Ub1 with allow_interface = b1.
- DomainParticipantQos.transports.enabled_transports = “Ub0”,”Ub1”
- DomainParticipantQos.discovery.enabled_transports = ”Ub0://”
- DomainParticipantQos.user_data.enabled_transports = ”Ub1://”

Node A sends a discovery packet to Node B to interface b0. This packet will be transformed using T0 as specified by Node A’s configuration. When this packet is received in Node B, it will be transformed using either T0 or T2 depending on the source address. Node’s A OS will use a0 or a1 to send this packet but *Connext DDS Micro* cannot ensure which one will be used. In case the OS sends the packet using a1, the wrong transformation will be applied in Node B.

Some systems have the possibility to configure the source address that should be used when a packet is sent. In POSIX systems, the command `ip route add <string> dev <interface>` can be used.

By typing the command `ip route add < b0 ip >/32 dev a0` in Node A, the OS will send all packets to Node B’s b0 IP address using interface a0. This would ensure that the correct transformation is applied in Node B. The same should be done to ensure that user data is sent with the right address `ip route add < b1 ip >/32 dev a1`. Of course, similar configuration is needed in Node B.

4.7 Discovery

This section discusses the implementation of discovery plugins in *RTI Connext DDS Micro*. For a general overview of discovery in *RTI Connext DDS Micro*, see *What is Discovery?*.

Connext DDS Micro discovery traffic is conducted through transports. Please see the *Transports* section for more information about registering and configuring transports.

4.7.1 What is Discovery?

Discovery is the behind-the-scenes way in which *RTI Connext DDS Micro* objects (*DomainParticipants*, *DataWriters*, and *DataReaders*) on different nodes find out about each other. Each *DomainParticipant* maintains a database of information about all the active *DataReaders* and *DataWriters* that are in the same DDS domain. This database is what makes it possible for *DataWriters* and *DataReaders* to communicate. To create and refresh the database, each application follows a common discovery process.

This section describes the default discovery mechanism known as the Simple Discovery Protocol, which includes two phases: *Simple Participant Discovery* and *Simple Endpoint Discovery*.

The goal of these two phases is to build, for each *DomainParticipant*, a complete picture of all the entities that belong to the remote participants that are in its peers list. The peers list is the list of nodes with which a participant may communicate. It starts out the same as the *initial_peers* list that you configure in the DISCOVERY QosPolicy. If the accept_unknown_peers flag in that same QosPolicy is TRUE, then other nodes may also be added as they are discovered; if it is FALSE, then the peers list will match the initial_peers list, plus any peers added using the *DomainParticipant's* **add_peer()** operation.

The following section discusses how *Connext DDS Micro* objects on different nodes find out about each other using the default Simple Discovery Protocol (SDP). It describes the sequence of messages that are passed between *Connext DDS Micro* on the sending and receiving sides.

The discovery process occurs automatically, so you do not have to implement any special code. For more information about advanced topics related to Discovery, please refer to the Discovery chapter in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access).

Simple Participant Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Participant Discovery Protocol (SPDP).

During the Participant Discovery phase, *DomainParticipants* learn about each other. The *DomainParticipant*'s details are communicated to all other *DomainParticipants* in the same DDS domain by sending participant declaration messages, also known as participant *DATA* submessages. The details include the *DomainParticipant*'s unique identifying key (GUID or Globally Unique ID described below), transport locators (addresses and port numbers), and QoS. These messages are sent on a periodic basis using best-effort communication.

Participant DATAs are sent periodically to maintain the liveliness of the *DomainParticipant*. They are also used to communicate changes in the *DomainParticipant*'s QoS. Only changes to QosPolicies that are part of the *DomainParticipant*'s built-in data need to be propagated.

When receiving remote participant discovery information, *RTI Connext DDS Micro* determines if the local participant matches the remote one. A 'match' between the local and remote participant occurs only if the local and remote participant have the same Domain ID and Domain Tag. This matching process occurs as soon as the local participant receives discovery information from the remote one. If there is no match, the discovery DATA is ignored, resulting in the remote participant (and all its associated entities) not being discovered.

When a *DomainParticipant* is deleted, a participant *DATA (delete)* submessage with the *DomainParticipant*'s identifying GUID is sent.

The GUID is a unique reference to an entity. It is composed of a GUID prefix and an Entity ID. By default, the GUID prefix is calculated from the IP address and the process ID. The entityID is set by *Connext DDS Micro* (you may be able to change it in a future version).

Once a pair of remote participants have discovered each other, they can move on to the Endpoint Discovery phase, which is how *DataWriters* and *DataReaders* find each other.

Simple Endpoint Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Endpoint Discovery Protocol (SEDP).

During the Endpoint Discovery phase, *RTI Connext DDS Micro* matches *DataWriters* and *DataReaders*. Information (GUID, QoS, etc.) about your application's *DataReaders* and *DataWriters* is exchanged by sending publication/subscription declarations in DATA messages that we will refer to as *publication DATAs* and *subscription DATAs*. The Endpoint Discovery phase uses reliable communication.

These declaration or DATA messages are exchanged until each *DomainParticipant* has a complete database of information about the participants in its peers list and their entities. Then the discovery process is complete and the system switches to a steady state. During steady state, *participant DATAs* are still sent periodically to maintain the liveliness status of participants. They may also be sent to communicate QoS changes or the deletion of a *DomainParticipant*.

When a remote *DataWriter/DataReader* is discovered, *Connext DDS Micro* determines if the local application has a matching *DataReader/DataWriter*. A 'match' between the local and remote entities occurs only if the *DataReader* and *DataWriter* have the same *Topic*, same data type, and compatible QosPolicies. Furthermore, if the *DomainParticipant* has been set up to ignore certain *DataWriters/DataReaders*, those entities will not be considered during the matching process.

This 'matching' process occurs as soon as a remote entity is discovered, even if the entire database is not yet complete: that is, the application may still be discovering other remote entities.

A *DataReader* and *DataWriter* can only communicate with each other if each one's application has hooked up its local entity with the matching remote entity. That is, both sides must agree to the connection.

Please refer to the section on Discovery Implementation in the *RTI Connext DDS Core Libraries User's Manual* for more details about the discovery process (available here if you have Internet access).

4.7.2 Configuring Participant Discovery Peers

An *RTI Connext DDS Micro DomainParticipant* must be able to send participant discovery announcement messages for other *DomainParticipants* to discover itself, and it must receive announcements from other *DomainParticipants* to discover them.

To do so, each *DomainParticipant* will send its discovery announcements to a set of locators known as its peer list, where a peer is the transport locator of one or more potential other *DomainParticipants* to discover.

peer_desc_string

A peer descriptor string of the initial_peers string sequence conveys the interface and address of the locator to which to send, as well as the indices of participants to which to send. For example:

```
DDS_StringSeq_set_maximum(&dp_qos.discovery.initial_peers, 3);
DDS_StringSeq_set_length(&dp_qos.discovery.initial_peers, 3);

*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 0) =
     DDS_String_dup("_udp://239.255.0.1");

*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 1) =
     DDS_String_dup("[1-4]@_udp://10.10.30.101");

*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 2) =
     DDS_String_dup("[2]@_udp://10.10.30.102");
```

The peer descriptor format is:

```
[index@][interface://]address
```

Remember that every *DomainParticipant* has a participant index that is unique within a DDS domain. The participant index (also referred to as the participant ID), together with the DDS domain ID, is used to calculate the network port on which *DataReaders* of that participant will receive messages. Thus, by specifying the participant index, or a range of indices, for a peer locator, that locator becomes a port to which messages will be sent only if addressed to the entities of a particular *DomainParticipant*. Specifying indices restricts the number of participant announcements sent to a locator where other *DomainParticipants* exist and, thus, should be considered to minimize network bandwidth usage.

In the above example, the first peer, "_udp://239.255.0.1," has the default UDPv4 multicast peer locator. Note that there is no [index@] associated with a multicast locator.

The second peer, "[1-4]@_udp://10.10.30.101," has a unicast address. It also has indices in brackets, [1-4]. These represent a range of participant indices, 1 through 4, to which participant discovery messages will be sent.

Lastly, the third peer, "[2]@_udp://10.10.30.102," is a unicast locator to a single participant with index 2.

4.7.3 Configuring Initial Peers and Adding Peers

DiscoveryQosPolicy_initial_peers is the list of peers a *DomainParticipant* sends its participant announcement messages, when it is enabled, as part of the discovery process.

DiscoveryQosPolicy_initial_peers is an empty sequence by default, so while DiscoveryQosPolicy_enabled_transports by default includes the DDS default loopback and multicast (239.255.0.1) addresses, initial_peers must be configured to include them.

Peers can also be added to the list, before and after a *DomainParticipant* has been enabled, by using DomainParticipant_add_peer.

The *DomainParticipant* will start sending participant announcement messages to the new peer as soon as it is enabled.

4.7.4 Discovery Plugins

When a *DomainParticipant* receives a participant discovery message from another *DomainParticipant*, it will engage in the process of exchanging information of user-created *DataWriter* and *DataReader* endpoints.

RTI Connext DDS Micro provides two ways of determinig endpoint information of other *DomainParticipants*: *Dynamic Discovery Plugin* and *Static Discovery Plugin*.

Dynamic Discovery Plugin

Dynamic endpoint discovery uses builtin discovery *DataWriters* and *DataReader* to exchange messages about user created *DataWriter* and *DataReaders*. A *DomainParticipant* using dynamic participant, dynamic endpoint (DPDE) discovery will have a pair of builtin *DataWriters* for sending messages about its own user created *DataWriters* and *DataReaders*, and a pair of builtin *DataReaders* for receiving messages from other *DomainParticipants* about their user created *DataWriters* and *DataReaders*.

Given a *DomainParticipant* with a user *DataWriter*, receiving an endpoint discovery message for a user *DataReader* allows the *DomainParticipant* to get the type, topic, and QoS of the *DataReader* that determine whether the *DataReader* is a match. When a matching *DataReader* is discovered, the *DataWriter* will include that *DataReader* and its locators as destinations for its subsequent writes.

Static Discovery Plugin

Static endpoint discovery uses function calls to statically assert information about remote endpoints belonging to remote *DomainParticipants*. An application with a *DomainParticipant* using dynamic participant, static endpoint (DPSE) discovery has control over which endpoints belonging to particular remote *DomainParticipants* are discoverable.

Whereas dynamic endpoint-discovery can establish matches for all endpoint-discovery messages it receives, static endpoint-discovery establishes matches only for the endpoint that have been asserted programmatically.

With DPSE, a user needs to know *a priori* the configuration of the entities that will need to be discovered by its application. The user must know the names of all *DomainParticipants* within the DDS domain and the exact QoS of the remote *DataWriters* and *DataReaders*.

Please refer to the C API Reference and C++ API Reference for the following remote entity assertion APIs:

- DPSE_RemoteParticipant_assert
- DPSE_RemotePublication_assert
- DPSE_RemoteSubscription_assert

Remote Participant Assertion

Given a local *DomainParticipant*, static discovery requires first the names of remote *DomainParticipants* to be asserted, in order for endpoints on them to match. This is done by calling DPSE_RemoteParticipant_assert with the name of a remote *DomainParticipant*. The name must match the name contained in the participant discovery announcement produced by that *DomainParticipant*. This has to be done reciprocally between two *DomainParticipants* so that they may discover one another.

For example, a *DomainParticipant* has entity name "participant_1", while another *DomainParticipant* has name "participant_2." participant_1 should call DPSE_RemoteParticipant_assert("participant_2") in order to discover participant_2. Similarly, participant_2 must also assert participant_1 for discovery between the two to succeed.

```
/* participant_1 is asserting (remote) participant_2 */
retcode = DPSE_RemoteParticipant_assert(participant_1,
                                        "participant_2");
if (retcode != DDS_RETCODE_OK) {
    printf("participant_1 failed to assert participant_2\n");
    goto done;
}
```

Remote Publication and Subscription Assertion

Next, a *DomainParticipant* needs to assert the remote endpoints it wants to match that belong to an already asserted remote *DomainParticipant*. The endpoint assertion function is used, specifying an argument which contains all the QoS and configuration of the remote endpoint. Where DPDE gets remote endpoint QoS information from received endpoint-discovery messages, in DPSE, the remote endpoint's QoS must be configured locally. With remote endpoints asserted, the *DomainParticipant* then waits until it receives a participant discovery announcement from an asserted remote *DomainParticipant*. Once received that, all endpoints that have been asserted for that remote *DomainParticipant* are considered discovered and ready to be matched with local endpoints.

Assume participant_1 contains a *DataWriter*, and participant_2 has a *DataReader*, both communicating on topic HelloWorld. participant_1 needs to assert the *DataReader* in participant_2 as a remote subscription. The remote subscription data passed to the operation must match exactly the QoS actually used by the remote *DataReader*:

```
/* Set participant_2's reader's QoS in remote subscription data  */
rem_subscription_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 200;
rem_subscription_data.topic_name = DDS_String_dup("Example HelloWorld");
rem_subscription_data.type_name = DDS_String_dup("HelloWorld");
rem_subscription_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
```

(continues on next page)

(continued from previous page)

```
/* Assert reader as a remote subscription belonging to (remote) participant_2 */
retcode = DPSE_RemoteSubscription_assert(participant_1,
                                         "participant_2",
                                         &rem_subscription_data,
                                         HelloWorld_get_key_kind(HelloWorldTypePlugin_
↪get(), NULL));
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to assert remote subscription\n");
    goto done;
}
```

Reciprocally, participant_2 must assert participant_1's *DataWriter* as a remote publication, also specifying matching QoS parameters:

```
/* Set participant_1's writer's QoS in remote publication data  */
rem_publication_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 100;
rem_publication_data.key.value.topic_name = DDS_String_dup("Example HelloWorld");
rem_publication_data.key.value.type_name = DDS_String_dup("HelloWorld");
rem_publication_data.key.value.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

/* Assert writer as a remote publication belonging to (remote) participant_1 */
retcode = DPSE_RemotePublication_assert(participant_2,
                                        "participant_1",
                                        &rem_publication_data,
                                        HelloWorld_get_key_kind(HelloWorldTypePlugin_
↪get(), NULL));
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to assert remote publication\n");
    goto done;
}
```

When participant_1 receives a participant discovery message from participant_2, it is aware of participant_2, based on its previous assertion, and it knows participant_2 has a matching *DataReader*, also based on the previous assertion of the remote endpoint. It therefore establishes a match between its *DataWriter* and participant_2's *DataReader*. Likewise, participant_2 will match participant_1's *DataWriter* with its local *DataRead*, upon receiving one of participant_1's participant discovery messages.

Note, with DPSE, there is no runtime check of QoS consistency between *DataWriters* and *DataReaders*, because no endpoint discovery messages are exchanged. This makes it extremely important that users of DPSE ensure that the QoS set for a local *DataWriter* and *DataReader* is the same QoS being used by another *DomainParticipant* to assert it as a remote *DataWriter* or *DataReader*.

4.8 Generating Type Support with rtiddsgen

4.8.1 Why Use rtiddsgen?

For *Connext DDS Micro* to publish and subscribe to topics of user-defined types, the types have to be defined and programmatically registered with *Connext DDS Micro*. A registered type is then serialized and deserialized by *Connext DDS Micro* through a pluggable type interface that each type must implement.

Rather than have users manually implement each new type, *Connext DDS Micro* provides the *rtiddsgen* utility for automatically generating type support code.

4.8.2 IDL Type Definition

rtiddsgen for *Connext DDS Micro* accepts types defined in IDL. The HelloWorld examples included with *Connext DDS Micro* use the following HelloWorld.idl:

```
struct HelloWorld {
    string<128> msg;
};
```

For further reference, see the section on Creating User Data Types with IDL in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access).

4.8.3 Generating Type Support

Before running *rtiddsgen*, some environment variables must be set:

- `RTIMEHOME` sets the path of the *Connext DDS Micro* installation directory
- `RTIMEARCH` sets the platform architecture (e.g. i86Linux2.6gcc4.4.5 or i86Win32VS2010)
- `JREHOME` sets the path for a Java JRE

Note that a JRE is shipped with *Connext DDS Micro* on platforms supported for the execution of rtiddsgen (Linux, Windows, and Mac® OS X®). It is not necessary to set `JREHOME` on these platforms, unless a specific JRE is preferred.

C

Run *rtiddsgen* from the command line to generate C language type-support for a UserType.idl (and replace any existing generated files):

```
> $rti_connext_micro_root/rtiddsgen/scripts/rtiddsgen -micro -language C -replace␣
↪UserType.idl
```

C++

Run *rtiddsgen* from the command line to generate C++ language type-support for a UserType.idl (and replace any existing generated files):

```
> $rti_connext_micro_root/rtiddsgen/scripts/rtiddsgen -micro -language C++ -replace␣
↪UserType.idl
```

Notes on Command-Line Options

In order to target *Connext DDS Micro* when generating code with *rtiddsgen*, the `-micro` option must be specified on the command line.

To list all command-line options specifically supported by *rtiddsgen* for *Connext DDS Micro*, enter:

```
> $rti_connext_micro_root/rtiddsgen/scripts/rtiddsgen -micro -help
```

Existing users might notice that that previously available options, `-language microC` and `-language microC++`, have been replaced by `-micro -language C` and `-micro -language C++`, respectively. It is still possible to specify `microC` and `microC++` for backwards compatibility, but users are advised to switch to using the `-micro` command-line option along with other arguments.

Generated Type Support Files

rtiddsgen will produce the following header and source files for each IDL file passed to it:

- UserType.h and UserType.c(xx) implement creation/intialization and deletion of a single sample and a sequence of samples of the type (or types) defined in the IDL description.
- UserTypePlugin.h and UserTypePlugin.c(xx) implement the pluggable type interface that *Connext DDS Micro* uses to serialize and deserialize the type.
- UserTypeSupport.h and UserTypeSupport.c(xx) define type-specific *DataWriters* and *DataReaders* for user-defined types.

4.8.4 Using custom data-types in Connext DDS Micro Applications

A *Connext DDS Micro* application must first of all include the generated headers. Then it must register the type with the *DomainParticipant* before a topic of that type can be defined. For an example HelloWorld type, the following code registers the type with the participant and then creates a topic of that type:

```
#include "HelloWorldPlugin.h"

/* ... */

retcode = DDS_DomainParticipant_register_type(application->participant,
                                              "HelloWorld",
                                              HelloWorldTypePlugin_get());
if (retcode != DDS_RETCODE_OK)
{
    /* Log an error */
    goto done;
}

application->topic =
```

(continues on next page)

(continued from previous page)

```
    DDS_DomainParticipant_create_topic(application->participant,
                                       "Example HelloWorld",
                                       "HelloWorld",
                                       &DDS_TOPIC_QOS_DEFAULT, NULL,
                                       DDS_STATUS_MASK_NONE);

if (application->topic == NULL)
{
    /* Log an error */
    goto done;
}
```

See the full HelloWorld examples for further reference.

4.8.5 Customizing generated code

rtiddsgen allows *Connext DDS Micro* users to select whether they want to generate code to subscribe to and/or publish a custom data-type. When generating code for subscriptions, only those parts of code dealing with deserialization of data and the implementation of a typed *DataReader* endpoint are generated. Conversely, only those parts of code addressing serialization and the implementation of a *DataWriter* are considered when generating publishing code.

Control over these options is provide by two command-line arguments:

- `-reader` generates code for deserializing custom data-types and creating *DataReaders* from them.
- `-writer` generates code for serializing custom data-types and creating *DataWriters* from them.

If neither of these two options are supplied to *rtiddsgen*, they will both be considered active and code for both *DataReaders* and *DataWriters* will be generated. If only one of the two options is supplied to *rtiddsgen*, only that one is enabled. If both options are supplied, both are enabled.

4.8.6 Unsupported Features of rtiddsgen with Connext DDS Micro

Connext DDS Micro supports a subset of the features and options in *rtiddsgen*. Use `rtiddsgen -micro -help` to see the list of features supported by *rtiddsgen* for *Connext DDS Micro*.

4.9 Threading Model

4.9.1 Introduction

This section describes the threading model, the use of critical sections, and how to configure thread parameters in *RTI Connext DDS Micro*. Please note that the information contained in this document applies to application development using *Connext DDS Micro*. For information regarding *porting* the *Connext DDS Micro* thread API to a new OS, please refer to *Porting RTI Connext DDS Micro*.

This section includes:

- *Architectural Overview*
- *Threading Model*
- *UDP Transport Threads*

4.9.2 Architectural Overview

RTI Connext DDS Micro consists of a core library and a number of components. The core library provides a porting layer, frequently used data-structures and abstractions, and the DDS API. Components provide additional functionality such as UDP communication, DDS discovery plugins, DDS history caches, etc.

```
+--------+                                  \
| DDS_C |                                    }  C API
+--------+                                  /

+--------+ +--------+ +-------+ +-------+         \
| DPSE   | | DPDE   | | WHSM | | RHSM |          |
+--------+ +--------+ +-------+ +-------+         |
+--------+ +--------+ +-------+ +-------+ +------+   } Optional components
| LOOP   | | UDP(*)| | RTPS | | DRI   | | DWI |   |  (platform independent)
+--------+ +--------+ +-------+ +-------+ +------+   |
                                                  /

+--------+ +--------+ +-------+ +-------+         \  Core Services (always
| REDA   | |  CDR   | | DB    | | RT    |          } present, platform
+--------+ +--------+ +-------+ +-------+         /  independent)

+------------------------------------+         \
|               OSAPI                |          } Platform dependent module
+------------------------------------+         /

(*) The UDP transport relies on a BSD socket API
```

4.9.3 Threading Model

RTI Connext DDS Micro is architected in a way that makes it possible to create a port of *Connext DDS Micro* that uses no threads, for example on platforms with no operating system. Thus, the following discussion can only be guaranteed to be true for *Connext DDS Micro* libraries from RTI.

OSAPI Threads

The *Connext DDS Micro* OSAPI layer creates one thread per OS process. This thread manages all the *Connext DDS Micro* timers, such as deadline and liveliness timers. This thread is created by the *Connext DDS Micro* OSAPI System when the OSAPI_System_initialize() function is called. When the *Connext DDS Micro* DDS API is used DomainParticipantFactory_get_instance() calls this function once.

Configuring OSAPI Threads

The timer thread is configured through the OSAPI_SystemProperty structure and any changes must be made before OSAPI_System_initialize() is called. In *Connext DDS Micro*, DomainParticipantFactory_get_instance() calls OSAPI_System_initialize(). Thus, if it is necessary to change the system timer thread settings, it must be done before DomainParticipantFactory_get_instance() is called the first time.

Please refer to OSAPI_Thread for supported thread options. Note that not all options are supported by all platforms.

```
struct OSAPI_SystemProperty sys_property = OSAPI_SystemProperty_INITIALIZER;

if (!OSAPI_System_get_property(&sys_property))
{
    /* ERROR */
}

/* Please refer to OSAPI_ThreadOptions for possible options */
sys.property.timer_property.thread.options = ....;

/* The stack-size is platform dependent, it is passed directly to the OS */
sys.property.timer_property.thread.stack_size = ....

/* The priority is platform dependent, it is passed directly to the OS */
sys.property.timer_property.thread.priority = ....

if (!OSAPI_System_set_property(&sys_property))
{
    /* ERROR */
}
```

UDP Transport Threads

Of the components that RTI provides, only the UDP component creates threads. The UDP transport creates one receive thread for each unique UDP receive address and port. Thus, three UDP threads are created by default:

- A multicast receive thread for discovery data (assuming multicast is available and enabled)
- A unicast receive thread for discovery data
- A unicast receive thread for user-data

Additional threads may be created depending on the transport configuration for a *DomainParticipant*, *DataReader* and *DataWriter*. The UDP transport creates threads based on the following criteria:

- Each unique unicast port creates a new thread
- Each unique multicast address *and* port creates a new thread

For example, if a *DataReader* specifies its own multicast receive address a new receive thread will be created.

Configuring UDP Receive Threads

All threads in the UDP transport share the same thread settings. It is important to note that all the UDP properties must be set before the UDP transport is registered. *Connext DDS Micro* pre-registers the UDP transport with default settings when the DomainParticipantFactory is initialized. To change the UDP thread settings, use the following code.

```
RT_Registry_T *registry = NULL;
DDS_DomainParticipantFactory *factory = NULL;
struct UDP_InterfaceFactoryProperty *udp_property = NULL;

factory = DDS_DomainParticipantFactory_get_instance();

udp_property = (struct UDP_InterfaceFactoryProperty *)
                    malloc(sizeof(struct UDP_InterfaceFactoryProperty));
*udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

registry = DDS_DomainParticipantFactory_get_registry(factory);

if (!RT_Registry_unregister(registry, "_udp", NULL, NULL))
{
    /* ERROR */
}

/* Please refer to OSAPI_ThreadOptions for possible options */
udp_property->recv_thread.options = ...;

/* The stack-size is platform dependent, it is passed directly to the OS */
udp_property->recv_thread.stack_size = ....

/* The priority is platform dependent, it is passed directly to the OS */
udp_property->recv_thread.priority = ....

if (!RT_Registry_register(registry, "_udp",
                          UDP_InterfaceFactory_get_interface(),
                          (struct RT_ComponentFactoryProperty*)udp_property,
                          NULL))
{
    /* ERROR */
}
```

General Thread Configuration

The *Connext DDS Micro* architecture consists of a number of components and layers, and each layer and component has its own properties. It is important to remember that the layers and components are configured independently of each other, as opposed to configuring everything through DDS. This design makes it possible to relatively easily swap out one part of the library for another.

All threads created based on *Connext DDS Micro* OSAPI APIs use the same OSAPI_ThreadProperty structure.

4.9.4 Critical Sections

RTI Connext DDS Micro may create multiple threads, but from an application point of view there is only a single critical section protecting all DDS resources. Note that although *Connext DDS Micro* may create multiple mutexes, these are used to protect resources in the OSAPI layer and are thus not relevant when using the public DDS APIs.

Calling DDS APIs from listeners

When DDS is executing in a listener, it holds a critical section. Thus it is important to return as quickly as possible to avoid stalling network I/O.

There are no deadlock scenarios when calling *Connext DDS Micro* DDS APIs from a listener. However, there are no checks on whether or not an API call will cause problems, such as deleting a participant when processing data in on_data_available from a reader within the same participant.

4.10 Batching

This section is organized as follows:

- *Overview*
- *Interoperability*
- *Performance*
- *Example Configuration*

4.10.1 Overview

Batching refers to a mechanism that allows *RTI Connext DDS Micro* to collect multiple user data DDS samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput.

Connext DDS Micro supports receiving batches of user data DDS samples, but does not support any mechanism to collect and send batches of user data.

Receiving batches of user samples is transparent to the application, which receives the samples as if the samples had been received one at a time. Note though that the reception sequence number refers to the sample sequence number, not the RTPS sequence number used to send RTPS messages. The RTPS sequence number is the batch sequence number for the entire batch.

A *Connext DDS Micro DataReader* can receive both batched and non-batched samples.

For a more detailed explanation, please refer to the BATCH QosPolicy section in the *RTI Connext DDS Core Libraries User's Manual* (available here if you have Internet access).

4.10.2 Interoperability

RTI Connext DDS Professional supports both sending and receiving batches, whereas *RTI Connext DDS Micro* supports only receiving batches. Thus, this feature primarily exists in *Connext DDS Micro* to interoperate with *RTI Connext DDS* applications that have enabled batching. An *Connext DDS Micro DataReader* can receive both batched and non-batched samples.

4.10.3 Performance

The purpose of batching is to increase throughput when writing small DDS samples at a high rate. In such cases, throughput can be increased several-fold, approaching much more closely the physical limitations of the underlying network transport.

However, collecting DDS samples into a batch implies that they are not sent on the network immediately when the application writes them; this can potentially increase latency. But, if the application sends data faster than the network can support, an increased proportion of the network's available bandwidth will be spent on acknowledgements and DDS sample resends. In this case, reducing that overhead by turning on batching could decrease latency while increasing throughput.

4.10.4 Example Configuration

This section includes several examples that explain how to enable batching in *RTI Connext DDS Professional*. For more detailed and advanced configuration, please refer to the *RTI Connext DDS Core Libraries User's Manual*.

- This configuration ensures that a batch will be sent with a maximum of 10 samples:

```
<datawriter_qos>
    <publication_name>
        <name>HelloWorldDataWriter</name>
    </publication_name>
    <batch>
        <enable>true</enable>
        <max_samples>10</max_samples>
    </batch>
</datawriter_qos>
```

- This configuration ensures that a batch is automatically flushed after the delay specified by max_flush_delay. The delay is measured from the time the first sample in the batch is written by the application:

```
<datawriter_qos>
    <publication_name>
        <name>HelloWorldDataWriter</name>
    </publication_name>
    <batch>
        <enable>true</enable>
        <max_flush_delay>
            <sec>1</sec>
            <nanosec>0</nanosec>
        </max_flush_delay>
    </batch>
</datawriter_qos>
```

- The following configuration ensures that a batch is flushed automatically when max_data_bytes is reached (in this example 8192).

```
<datawriter_qos>
    <publication_name>
```

(continues on next page)

(continued from previous page)

```
        <name>HelloWorldDataWriter</name>
    </publication_name>
    <batch>
        <enable>true</enable>
        <max_data_bytes>8192</max_data_bytes>
    </batch>
</datawriter_qos>
```

Note that max_data_bytes does not include the metadata associated with the batch samples.

Batches must contain whole samples. If a new batch is started and its initial sample causes the serialized size to exceed max_data_bytes, *RTI Connext DDS Professional* will send the sample in a single batch.

4.11 Building Against FACE Conformance Libraries

This section describes how to build *Connext DDS Micro* using the FACE™ conformance test tools.

4.11.1 Requirements

Connext DDS Micro Source Code

The *Connext DDS Micro* source code is available from RTI's Support portal.

FACE Conformance Tools

RTI does not distribute the FACE conformance tools.

CMake

The *Connext DDS Micro* source is distributed with a **CMakeList.txt** project file. CMake is an easy to use tool that generates makefiles or project files for various build-tools, such has UNIX makefiles, Microsoft® Visual Studio® project files, and Xcode.

CMake can be downloaded from https://www.cmake.org.

4.11.2 FACE Golden Libraries

The FACE conformance tools use a set of golden libraries. There are different golden libraries for different FACE services, languages and profiles. *Connext DDS Micro only* conforms to the safetyExt and safety profile of OSS using the C language.

Building the FACE Golden Libraries

The FACE conformance tools ship with their own set of tools to build the golden libraries. Please follow the instructions provided by FACE. In order to build the FACE golden libraries, it is necessary to port to the required platform. RTI has only tested *Connext DDS Micro* on Linux 2.6 systems with GCC 4.4.5. The complete list of files modified by RTI are included below in source form.

4.11.3 Building the Connext DDS Micro Source

The following instructions show how to built the *Connext DDS Micro* source:

- Extract the source-code. Please note that the remaining instructions assume that only a single platform is built from the source.
- In the top-level source directory, enter the following:

```
shell> cmake-gui .
```

  This will start the CMake GUI where all build configuration takes place.

- Click the "Configure" button.
- Select UNIX Makefiles from the drop-down list.
- Select "Use default compilers" or "Specify native compilers" as required. Press "Done."
- Click "Configure" again. There should not be any red lines. If there are, click "Configure" again.

  NOTE: A red line means that a variable has not been configured. Some options could add new variables. Thus, if you change an option a new red lines may appear. In this case configure the variable and press "Configure."

- Expand the CMAKE and RTIMICRO options and configure how to build *Connext DDS Micro*:

```
CMAKE_BUILD_TYPE: Debug or blank. If Debug is used, the |me| debug
                  libraries are built.

RTIMICRO_BUILD_API: C or C++
   C   - Include the C API. For FACE, only C is supported.
   C++ - Include the C++ API.

RTIMICRO_BUILD_DISCOVERY_MODULE: Dynamic | Static | Both
   Dynamic - Include the dynamic discovery module.
   Static  - Include the static discovery module.
   Both    - Include both discovery modules.

RTIMICRO_BUILD_LIBRARY_BUILD:
   Single    - Build a single library.
   RTI style - Build the same libraries RTI normally ships. This is useful
               if RTI libraries are already being used and you want to use
               the libraries built from source.

RTIMICRO_BUILD_LIBRARY_TYPE:
    Static -  Build static libraries.
    Shared -  Build shared libraries.

RTIMICRO_BUILD_LIBRARY_PLATFORM_MODULE: POSIX

RTIMICRO_BUILD_LIBRARY_TARGET_NAME: <target name>
   Enter a string as the name of the target. This is also used as the
```

(continues on next page)

(continued from previous page)

```
   name of the directory where the built libraries are placed.
   If you are building libraries to replace the libraries shipped by RTI,
   you can use the RTI target name here. It is then possible to set
   RTIMEHOME to the source tree (if RTI style is selected for
   RTIMICRO_BUILD_LIBRARY_BUILD).

RTIMICRO_BUILD_ENABLE_FACE_COMPLIANCE: Select level of FACE compliance
    None            - No compliance required
    General         - Build for compliance with the FACE general profile
    Safety Extended - Build for compliance with the FACE safety extended profile
    Safety          - Build for compliance with the FACE safety profile

RTIMICRO_BUILD_LINK_FACE_GOLDEBLIBS:
    Check if linking against the static FACE conformance test libraries.
    NOTE: This check-box is only available if FACE compliance is different
    from "None".

RTIMICRO_BUILD_LINK_FACE_GOLDEBLIBS:
    If the  RTIMICRO_BUILD_LINK_FACE_GOLDEBLIBS is checked the path to the
    top-level FACE root must be specified here.
```

- Click “Configure”.
- Click “Generate”.
- Build the generated project.
- Libraries are placed in **lib/<RTIMICRO_BUILD_LIBRARY_TARGET_NAME>**.

4.12 Working With Sequences

4.12.1 Introduction

RTI Connext DDS Micro uses IDL as the language to define data-types. One of the constructs in IDL is the *sequence*: a variable-length vector where each element is of the same type. This section describes how to work with sequences; in particular, the string sequence since it has special properties.

4.12.2 Working with Sequences

Overview

Logically a sequence can be viewed as a variable-length vector with N elements, as illustrated below. Note that sequences indices are 0 based.

```
   +---+
0  | T |
   +---+
1  | T |
   +---+
```

(continues on next page)

(continued from previous page)

```
  2  | T |
     +---+
       |
       |
     +---+
N-1  | T |
     +---+
```

There are three types of sequences in *Connext DDS Micro*:

- Builtin sequences of primitive IDL types.
- Sequences defined in IDL using the sequence keyword.
- Sequences defined by the application.

The following builtin sequences exist (please refer to C API Reference and C++ API Reference for the complete API).

IDL Type	*Connext DDS Micro* Type	*Connext DDS Micro* Sequence
octet	DDS_Octet	DDS_OctetSeq
char	DDS_Char	DDS_CharSeq
boolean	DDS_Boolean	DDS_BooleanSeq
short	DDS_Short	DDS_ShortSeq
unsigned short	DDS_UnsignedShort	DDS_UnsignedShortSeq
long	DDS_Long	DDS_LongSeq
unsigned long	DDS_UnsignedLong	DDS_UnsignedLongSeq
enum	DDS_Enum	DDS_EnumSeq
wchar	DDS_Wchar	DDS_WcharSeq
long long	DDS_LongLong	DDS_LongLongSeq
unsigned long long	DDS_UnsignedLongLong	DDS_UnsignedLongLongSeq
float	DDS_Float	DDS_FloatSeq
double	DDS_Double	DDS_DoubleSeq
long double	DDS_LongDouble	DDS_LongDoubleSeq
string	DDS_String	DDS_StringSeq
wstring	DDS_Wstring	DDS_WstringSeq

The following are important properties of sequences to remember:

- All sequences in *Connext DDS Micro* *must* be finite.
- All sequences defined in IDL are sized based on IDL properties and *must* not be resized. That is, *never* call **set_maximum()** on a sequence defined in IDL. This is particularly important for string sequences.
- Application defined sequences can be resized using **set_maximum()** or **ensure_length()**.
- There are two ways to use a **DDS_StringSeq** (they are type-compatible):
  - A **DDS_StringSeq** originating from IDL. This sequence is sized based on maximum sequence length *and* maximum string length.

   - A **DDS_StringSeq** originating from an application. In this case the sequence element memory is unmanaged.

- All sequences have an initial length of 0.

Working with IDL Sequences

Sequences that originate from IDL are created when the IDL type they belong to is created. IDL sequences are always initialized with the maximum size specified in the IDL file. The maximum size of a type, and hence the sequence size, is used to calculate memory needs for serialization and deserialization buffers. Thus, changing the size of an IDL sequence can lead to hard to find memory corruption.

The string and wstring sequences are special in that not only is the maximum sequence size allocated, but because strings are also always of a finite maximum length, the maximum space needed for each string element is also allocated. This ensure that *Connext DDS Micro* can prevent memory overruns and validate input.

Some typical scenarios with a long sequence and a string sequence defined in IDL is shown below:

```
/* In IDL */
struct SomeIdlType
{
    // A sequence of 20 longs
    sequence<long,20> long_seq;

    // A sequence of 10 strings, each string has a maximum length of 255 bytes
    // (excluding NUL)
    sequence<string<255>,10> string_seq;
}

/* In C source */
SomeIdlType *my_sample = SomeIdlTypeTypeSupport_create_data()

DDS_LongSet_set_length(&my_sample->long_seq,5);
DDS_StringSeq_set_length(&my_sample->string_seq,5);

/* Assign the first 5 longs in long_seq */
for (i = 0; i < 5; ++i)
{
    *DDS_LongSeq_get_reference(&my_sample->long_seq,i) = i;
    snprintf(*DDS_StringSeq_get_reference(&my_sample->string_seq,0),255,"SomeString %d",
↪i);
}

SomeIdlTypeTypeSupport_delete_data(my_sample);

/* In C++ source */
SomeIdlType *my_sample = SomeIdlTypeTypeSupport::create_data()

/* Assign the first 5 longs in long_seq */

my_sample->long_seq.length(5);
```

(continues on next page)

(continued from previous page)

```
my_sample->string_seq.length(5);

for (i = 0; i < 5; ++i)
{
    /* use method */
    *DDSLongSeq_get_reference(&my_sample->long_seq,i) = i;
    snprintf(*DDSStringSeq_get_reference(&my_sample->string_seq,i),255,"SomeString %d",
↪i);

    /* or assignment */
    my_sample->long_seq[i] = i;
    snprintf(my_sample->string_seq[i],255,"SomeString %d",i);
}

SomeIdlTypeTypeSupport::delete_data(my_sample);
```

Note that in the example above the sequence length is set. The maximum size for each sequence is set when my_sample is allocated.

A special case is to copy a string sequence from a sample to a string sequence defined outside of the sample. This is possible, but care *must* be taken to ensure that the memory is allocated properly:

Consider the IDL type from the previous example. A string sequence of equal size can be allocated as follows:

```
struct DDS_StringSeq app_seq = DDS_SEQUUENCE_INITIALIZER;

/* This ensures that memory for the strings are allocated upfront */
DDS_StringSeq_set_maximum_w_max(&app_seq,10,255);

DDS_StringSeq_copy(&app_seq,&my_sample->string_seq);
```

If instead the following code was used, memory for the string in **app_seq** would be allocated as needed.

```
struct DDS_StringSeq app_seq = DDS_SEQUUENCE_INITIALIZER;

/* This ensures that memory for the strings are allocated upfront */
DDS_StringSeq_set_maximum(&app_seq,10);

DDS_StringSeq_copy(&app_seq,&my_sample->string_seq);
```

Working with Application Defined Sequences

Application defined sequences work in the same way as sequences defined in IDL with two exceptions:

- The maximum size is 0 by default. It is necessary to call **set_maximum** or ensure_length to allocate space.
- **DDS_StringSet_set_maximum** does not allocate space for the string pointers. The memory must be allocated on a per needed basis and calls to **_copy** may reallocate memory

as needed. Use **DDS_StringSeq_set_maximum_w_max** or **DDS_StringSeq_ensure_length_w_max** to also allocate pointers. In this case **_copy** will *not* reallocate memory.

Note that it is not allowed to mix the use of calls that pass the max (ends in **_w_max**) and calls that do not. Doing so may cause memory leaks and/or memory corruption.

```
struct DDS_StringSeq my_seq = DDS_SEQUENCE_INITIALIZER;

DDS_StringSeq_ensure_length(&my_seq,10,20);

for (i = 0; i < 10; i++)
{
    *DDS_StringSeq_get_reference(&my_seq,i) = DDS_String_dup("test");
}

DDS_StringSeq_finalize(&my_seq);
```

DDS_StringSeq_finalize automatically frees memory pointed to by each element using **DDS_String_free**. All memory allocated to a string element should be allocated using a **DDS_String** function.

It is possible to assign any memory to a string sequence element if all elements are released manually first:

```
struct DDS_StringSeq my_seq = DDS_SEQUENCE_INITIALIZER;

DDS_StringSeq_ensure_length(&my_seq,10,20);

for (i = 0; i < 10; i++)
{
    *DDS_StringSeq_get_reference(&my_seq,i) = static_string[i];
}

/* Work with the sequence */

for (i = 0; i < 10; i++)
{
    *DDS_StringSeq_get_reference(&my_seq,i) = NULL;
}

DDS_StringSeq_finalize(&my_seq);
```

4.13 Debugging

4.13.1 Overview

Connext DDS Micro maintains a log of events occuring in a *Connext DDS Micro* application. Information on each event is formatted into a log entry. Each entry can be stored in a buffer, stringified into a displayable log message, and/or redirected to a user-defined log handler.

For a list of error codes, please refer to C Logging Reference or C++ Logging Reference

4.13.2 Configuring Logging

By default, *Connext DDS Micro* sets the log verbosity to *Error*. It can be changed at any time by calling **OSAPI_Log_set_verbosity()** using the desired verbosity as a parameter.

Note that when compiling with RTI_CERT defined, logging is completely removed.

The *Connext DDS Micro* log stores new log entries in a log buffer.

The default buffer size is different for Debug and Release libraries. The Debug libraries are configured to use a much larger buffer than the Release ones. A custom buffer size can be configured using the **OSAPI_Log_set_property()** function. For example, to set a buffer size of 128 bytes:

```
struct OSAPI_LogProperty prop = OSAPI_LogProperty_INIITALIZER;

OSAPI_Log_get_property(&prop);
prop.max_buffer_size = 128;
OSAPI_Log_set_property(&prop);
```

Note that if the buffer size is too small, log entries will be truncated in order to fit in the available buffer.

The function used to write the logs can be set during compilation by defining the macro OSAPI_LOG_WRITE_BUFFER. This macro shall have the same parameters as the function prototype **OSAPI_Log_write_buffer_T**.

It is also possible to set this function during runtime by using the function **OSAPI_Log_set_property()**:

```
struct OSAPI_LogProperty prop = OSAPI_LogProperty_INIITALIZER;

OSAPI_Log_get_property(&prop);
prop.write_buffer = <pointer to user defined write function>;
OSAPI_Log_set_property(&prop);
```

A user can install a log handler function to process each new log entry. The handler must conform to the definition OSAPI_LogHandler_T, and it is set by **OSAPI_Log_set_log_handler()**.

When called, the handler has parameters containing the raw log entry and detailed log information (e.g., error code, module, file and function names, line number).

The log handler is called for every new log entry, even when the log buffer is full. An expected use case is redirecting log entries to another logger, such as one native to a particular platform.

4.13.3 Log Message Kinds

Each log entry is classified as one of the following kinds:

- *Error*. An unexpected event with negative functional impact.
- *Warning*. An event that may not have negative functional impact but could indicate an unexpected situation.
- *Information*. An event logged for informative purposes.

By default, the log verbosity is set to *Error*, so only error logs will be visible. To change the log verbosity, simply call the function **OSAPI_Log_set_verbosity()** with the desired verbosity level.

4.13.4 Interpreting Log Messages and Error Codes

A log entry in *Connext DDS Micro* has a defined format.

Each entry contains a header with the following information:

- *Length*. The length of the log message, in bytes.
- *Module ID*. A numerical ID of the module from which the message was logged.
- *Error Code*. A numerical ID for the log message. It is unique within a module.

Though referred to as an "error" code, it exists for all log kinds (error, warning, info).

The module ID and error code together uniquely identify a log message within *Connext DDS Micro*.

Connext DDS Micro can be configured to provide additional details per log message:

- *Line Number*. The line number of the source file from which the message is logged.
- *Module Name*. The name of the module from which the message is logged.
- *Function Name*. The name of the function from which the message is logged.

When an event is logged, by default it is printed as a message to standard output. An example error entry looks like this:

```
[943921909.645099999]ERROR: ModuleID=7 Errcode=200 X=1 E=0 T=1
dds_c/DomainFactory.c:163/DDS_DomainParticipantFactory_get_instance: kind=19
```

- *X* Extended debug information is present, such as file and line number.
- *E* Exception, the log message has been truncated.
- *T* The log message has a valid timestamp (successful call to OSAPI_System_get_time()).

A log message will need to be interpreted by the user when an error or warning has occurred and its cause needs to be determined, or the user has set a log handler and is processing each log message based on its contents.

A description of an error code printed in a log message can be determined by following these steps:

- Navigate to the module that corresponds to the Module ID, or the printed module name in the second line. In the above example, "ModuleID=7" corresponds to DDS.
- Search for the error code to find it in the list of the module's error codes. In the example above, with "Errcode=200," search for "200" to find the log message that has the value "(DDSC_LOG_BASE + 200)".

4.14 Connext DDS Micro Hardcoded Resource Limits

4.14.1 Introduction

Connext DDS Micro contains a few resource limits that are not configurable in a QoS policy or property. Note that not every single constant used in *Connext DDS Micro* is addressed. The focus is on resource limits that may prevent an application using *Connext DDS Micro* from behaving correctly. For example, the maximum number of participants that can be discovered on a node may impact an application. On the other hand, a resource limit that has no functional impact, for example the maximum length of the discovery plugin name, is not described in this document.

When a resource limit is exceeded an error message is logged. An explanation can be found in the documentation. Note that some resource limits may be exceeded when calling an API and others may be exceeded as part of processing incoming data. Thus, it may be necessary to look at log output to see the failure reason.

Although *Connext DDS Micro* can be compiled from source it is recommended to consult with RTI before making any changes to the hard coded limits.

4.14.2 Summary

Resource	Limit
Number of domain participants per OS process	8
Max topic name length	255
Max type name length	255
Max number of discovery plugins used by a domain participant	1
Max number of announced receive addresses for discovery data by a domain participant	4
Max number of announced receive addresses for user-data data by a domain participant	4
Max number of addresses that can be received (per meta-unicast, meta-multicast, user-unicast, user-multicast)	4

4.14.3 Operating Services API (OSAPI)

- The maximum number of object ids are 2^32-1
  - DDS objects require a unique object_id. The encoding dictated by the RTPS specification limits the number of DDS objects within a domain participant to 2^24.
  - User impact - None.
- The Maximum number of timers that can be created is 8
  - Each domain participant allocates 1 timer
    - User impact - The maximum number of domain participants in a single OS process is limited to 8. This limit is based on empirical data; only specialized applications such as tools typically use more than 2 domain participants.

4.14.4 DDS C API

- Maximum Topic name length - 255 (including NUL termination)

  - The limit is specified as 256 including NUL termination in the RTPS specification, refer to 9.6.2.2.2 in the RTPS specification (OMG formal/2009-01-05).
- Maximum Type name length - 255 (including NUL termination)
  - The limit is specified as 256 including NUL termination in the RTPS specification, refer to 9.6.2.2.2 in the RTPS specification (OMG formal/2009-01-05).
- Maximum number of matched data-writers (per data-reader) - 100,000,000
  - This limit determines how many data-writers each data-reader can match.
- Maximum number of matched data-readers (per data-writer) - 100,000,000
  - This limit determines how many data-readers each data-writer can match.
- Maximum number of locators of each type which can be sent in the participant announcement - 4
  - This limit determines the number of unique network address that can be advertised as part of discovery. The limit is per locator type. That is, the limit is applicable to discovery and user-data (total of 4 each)
- Maximum number of discovery plugins which can be used by the domain participant - 1
  - User impact: Must choose either static or dynamic discovery.

4.14.5 Dynamic Discovery Plugin (DPDE)

- Maximum number of received locators - 4
  - This limit determines the number of unique network address that can be advertised as part of discovery.
  - The limit is per locator type. That is, the same limit is applicable to discovery unicast, discovery multicast, user-data unicast, and user-data multicast.

4.14.6 Static Discovery Plugin (DPSE)

- Maximum number of received locators - 4
  - This limit determines the number of unique network address that can be advertised as part of discovery.
  - The limit is per locator type. That is, the same limit is applicable to discovery unicast, discovery multicast, user-data unicast, and user-data multicast.

4.14.7 RTPS Protocol Implementation (RTPS)

- Unlimited max_samples is defined as 100000000
- Maximum number of external RTPS interfaces - 16
  - This limits the number of participants to 16 per OS process.
  - This limit is reduced to 8 due to the OS limit.

Chapter 5

Building and Porting

5.1 Connext DDS Micro Supported Platforms

RTI Connext DDS Micro is a source product and all platforms supported by RTI are supported. However, RTI does not test and validate the libraries on all permutations of CPU types, compiler version and OS version.

5.1.1 Reference Platforms

The following are reference platforms for which the platform-dependent layers provided with the *RTI Connext DDS Micro* product are tested as part of standard product release:

- Windows®
- Linux®
- Unix™ (POSIX Compliant)
- Wind River® VxWorks®
- Express Logic® ThreadX®
- FreeRTOS™
- macOS® X (Darwin)
- QNX® 6.6, 7

5.1.2 Known Customer Platforms

RTI Connext DDS Micro has been ported to a number of platforms by our customers, such as:

- uC/OS™
- uLinux
- Win32
- Android™
- iOS®

- TI's Stellaris® Arm® Cortex®-M3 and -M4 with only TI device drivers, no OS
- Baremetal - Arm Cortex-M4
- INTEGRITY®-178
- VxWorks 653 2.x, 3.x
- DDC-I Deos™
- LynxOS®-178
- VOS™

RTI Connext DDS Micro is known to run with the following network stacks: - BSD® socket-based stack - Windows Socket library - VxWorks Network stack - ThreadX Network stack - RTNet® - lwIP (event and blocking mode) - QNX Network stack - GHS IPFlite and general purpose stack

5.2 Building the Connext DDS Micro Source

5.2.1 Introduction

RTI Connext DDS Micro has been engineered for reasonable portability to common platforms and environments, such as Darwin, iOS, Linux, and Windows. This document explains how to build the *Connext DDS Micro* source-code. The focus of this document is building *Connext DDS Micro* for an architecture supported by RTI (please refer to *Connext DDS Micro Supported Platforms* for more information). Please refer to *Porting RTI Connext DDS Micro* for documentation on how to port *Connext DDS Micro* to an *unsupported* architecture.

This manual is written for developers and engineers with a background in software development. It is recommended to read the document in order, as one section may refer to or assume knowledge about concepts described in a preceding section.

5.2.2 The Host and Target Environment

The following terminology is used to refer to the environment in which *Connext DDS Micro* is built and run:

- The *host* is the machine that runs the software to compile and link *Connext DDS Micro*.
- The *target* is the machine that runs *Connext DDS Micro*.
- In many cases *Connext DDS Micro* is built *and* run on the same machine. This is referred to as a *self-hosted environment*.

The *environment* is the collection of tools, OS, compiler, linker, hardware etc. needed to build and run applications.

The word *must* describes a requirement that must be met. Failure to meet a *must* requirement may result in failure to compile, use or run *Connext DDS Micro*.

The word *should* describes a requirement that is strongly recommended to be met. A failure to meet a *should* recommendation may require modification to how *Connext DDS Micro* is built, used, or run.

The word *may* is used to describe an optional feature.

The Host Environment

RTI Connext DDS Micro has been designed to be easy to build and to require few tools on the host.

The host machine **must**:

- support long filenames (8.3 will not work). *Connext DDS Micro* does not require a case sensitive file-system.
- have the necessary compiler, linkers, and build-tools installed.

The host machine **should**:

- have CMake (www.cmake.org) installed. Note that it is not required to use CMake to build *Connext DDS Micro*, and in some cases it may also not be recommended. As a rule of thumb, if *RTI Connext DDS Micro* can be built from the command-line, CMake is recommended.
- be able to run bash shell scripts (Unix type systems) or BAT scripts (Windows machines).

Typical examples of host machines are:

- a Linux PC with the GNU tools installed (make, gcc, g++, etc).
- a Mac computer with Xcode and the command-line tools installed.
- a Windows computer with Microsoft Visual Studio Express edition.
- a Linux, Mac or Windows computer with an embedded development tool-suite.

The Target Environment

Connext DDS Micro has been designed to run on a wide variety of targets. For example, *Connext DDS Micro* can be ported to run with no OS, an RTOS, GNU libc or a non-standard C library etc. This section only lists the minimum requirements. Please refer to *Porting RTI Connext DDS Micro* for how to port *Connext DDS Micro*.

The target machine must:

- support 8, 16, and 32-bit signed and unsigned integer. Note that a 16 bit CPU (or even 8 bit) is supported as long as the listed types are supported.

  Connext DDS Micro supports 64 bit CPUs, and it does not use any native 64 bit quantities internally.

The target compiler should:

- have a C compiler that is C99 compliant. Note that many non-standard compilers work, but may require additional configuration.
- have a C++ compiler that is C++98 compliant.

The remainder of this manual assumes that the target environment is one supported by RTI:

- POSIX (Linux, Darwin, QNX®, VOS, iOS, Android).

- VxWorks 6.9 or later.
- Windows.
- QNX.

5.2.3 Overview of the Connext DDS Micro Source Bundle

The *Connext DDS Micro* source is available from the RTI support portal. If you do not have access, please contact RTI Support. The source-code is exactly the same as developed and tested by RTI. No filtering or modifications are performed, except for line-ending conversion for the Windows source bundle.

The source-bundle is in a directory called **src/** under your *Connext DDS Micro* installation.

```
RTIMEHOME--+-- CmakeLists.txt
           |
           +-- build -- cmake --+-- Debug --+-- <ARCH> -- <project-files>
           |                    |
           |                    |
           |                    +-- Release --+-- <ARCH> -- <project-files>
           +-- doc --
           |
           +-- example
           |
           +-- include
           |
           +-- lib +-- <ARCH> -- <libraries>
           |
           +-- resource --+-- cmake
           |              |
           |              +-- scripts
           |
           +-- rtiddsgen
           |
           +-- rtiddsmag
           |
           +-- src
```

In this document, `RTIMEHOME` refers to the root directory where RTI archives are extracted and installed. The only difference between the UNIX and Windows source bundles is the line endings.

Directory Structure

The recommended directory structure is described below and *should* be used (1) because:

- the source bundle includes a helper script to run CMake that expects this directory structure.
- this directory structure supports multiple architectures.
- this directory structure mirrors the structure shipped by RTI. (2).

NOTE 1: This applies to builds using CMake. To build in a custom environment, please refer to *Custom Build Environments*.

CMakeLists.txt is the main input file to CMake and is used to generate build files.

The *RTIMEHOME/include* directory contains the public header files. By default it is identical to *RTIMEHOME/include*. However, custom ports will typically add files to this directory.

The *RTIMEHOME/src* directory contains the *Connext DDS Micro* source files. RTI does not support modifications to these files unless explicitly stated in the porting guide. A custom port will typically add specific files to this directory.

The *RTIMEHOME/build* directory is empty by default. CMake generates one set of build-files for each configuration. A build configuration can be an architecture, *Connext DDS Micro* options, language selection, etc. This directory will contain CMake generated build-files per architecture per configuration. By convention the *Debug* directory is used to generate build-files for debug libraries and the *Release* directory is used for release libraries.

The *RTIMEHOME/lib* directory is empty by default. All libraries successfully built with the CMake generated build-files, regardless of which generator was used, will be copied to the *RTIMEHOME/lib* directory.

The following naming conventions are used regardless of the build-tool:

- Static libraries have a *z* suffix.
- Shared libraries do *not* have an additional suffix.
- Debug libraries have a *d* suffix.
- Release libraries do *not* have an additional suffix.

The following libraries are built:

- *rti_me* - the core library, including the DDS C API
- *rti_me_discdpde* - the Dynamic Participant Dynamic Endpoint plugin
- *rti_me_discdpse* - the Dynamic Participant Static Endpoint plugin
- *rti_me_rhsm* - the Reader History plugin
- *rti_me_whsm* - the Writer History plugin
- *rti_me_cpp* - the C++ API

Note: The names above are the RTI library names. Depending on the target architecture, the library name is prefixed with *lib* and the library suffix also varies between target architectures, such as .so, .dylib, etc.

For example:

- rti_mezd indicates a static debug library
- rti_me indicates a dynamically linked release library

5.2.4 Compiling Connext DDS Micro

This section describes in detail how to compile *Connext DDS Micro* using CMake. It starts with an example that uses the included scripts followed by a section showing how to build manually.

CMake, available from www.cmake.org, is the preferred tool to build *Connext DDS Micro* because it simplifies configuring the *Connext DDS Micro* build options and generates build files for a variety of environments. Note that CMake itself does not compile anything. CMake is used to *generate* build files for a number of environments, such as make, Eclipse® CDT, Xcode® and Visual Studio. Once the build-files have been generated, any of the tools mentioned can be used to build *Connext DDS Micro*. This system makes it easier to support building *Connext DDS Micro* in different build environments. CMake is easy to install with pre-built binaries for common environments and has no dependencies on external tools.

NOTE: It is not required to use CMake. Please refer to *Custom Build Environments* for other ways to build *Connext DDS Micro*.

Building Connext DDS Micro with rtime-make

The *Connext DDS Micro* source bundle includes a bash (UNIX) and BAT (Windows) script to simplify the invocation of CMake. These scripts are a convenient way to invoke CMake with the correct options.

On UNIX-based systems:

```
RTIMEHOME/resource/script/rtime-make --config Debug --target self \
                       --name i86Linux2.6gcc4.4.5 -G "Unix Makefiles" --build
```

On Windows systems:

```
RTIMEHOME\resource\scripts\rtime-make --config Debug --target self \
                    --name i86Win32VS2010 -G "Visual Studio 10 2010" --build
```

Explanation of arguments:

- `--config Debug` : Create Debug build.
- `--target \<target\>` : The target for the sources to be built. "self" indicates that the host machine is the target and *Connext DDS Micro* will be built with the options that CMake automatically determines for the local compiler. Please refer to *Cross-Compiling Connext DDS Micro* for information on specifying the target architecture to build for.
- `--name \<name\>` : The name of the build, shall be a descriptive name following the recommendation on naming described in section *Preparing for a Build*. If `--name` is not specified, the value for `--target` will be used as the name.
- `--build Build`: The generated project files.

On UNIX-based systems:

- If gcc is part of the name, GCC is assumed.
- If clang is part of the name, clang is assumed.

On Windows systems:

- If Win32 is part of the name, a 32 bit Windows build is assumed.
- If Win64 is part of the name, a 64 bit Windows build is assumed.

To get a list of all the options:

```
rtime-make -h
```

To get help for a specific target:

```
rtime-make --target <target> --help
```

Manually Building with CMake

Preparing for a Build

As mentioned, it is recommended to create a unique directory for each build configuration. A build configuration can be created to address specific architectures, compiler settings, or different *Connext DDS Micro* build options.

RTI recommends assigning a descriptive *name* to each build configuration, using a common format. While there are no requirements to the format for functional correctness, the tool-chain files in *Cross-Compiling Connext DDS Micro* uses the **RTIME_TARGET_NAME** variable to determine various compiler options and selections.

RTI uses the following name format:

```
{cpu}{OS}{compiler}_{config}
```

In order to avoid a naming conflict with RTI, the following name format is recommended:

```
{prefix}_{cpu}{OS}{compiler}_{config}
```

Some examples:

- acme_ppc604FreeRTOSgcc4.6.1 - *Connext DDS Micro* for a PPC 604 CPU running FreeRTOS compiled with gcc 4.6.1, compiled by acme.
- acme_i86Win32VS2015 - *Connext DDS Micro* for an i386 CPU running Windows XP or higher compiled with Visual Studio 2015, compiled by acme.
- acme_i86Linux4gcc4.4.5_test - a test configuration build of *Connext DDS Micro* for an i386 CPU running Linux 3 or higher compiled with gcc 4.4.5, compiled by acme.

Files built by each build configuration will be stored under *RTIMEHOME/build/[Debug | Release]/<name>*. These directories are referred to as build directories or `RTIMEBUILD`. The structure of the `RTIMEBUILD` depends on the generated build files and should be regarded as an intermediate directory.

Creating Build Files for Connext DDS Micro Using the CMake GUI

Start the CMake GUI, either from a terminal window or a menu.

Please note that the Cmake GUI does *not* set the **CMAKE_BUILD_TYPE** variable. This variable is used to determine the names of the *Connext DDS Micro* libraries. Thus, it is necessary to add **CMAKE_BUILD_TYPE** manually and specify either Debug or Release. To add this variable manually, click the 'Add Entry' button, specify the name as a string type.

As an alternative, rtime-make's `--gui` option can be used. This option starts the CMake and also adds the **CMAKE_BUILD_TYPE** option when the CMake GUI exits.

Please note that when using Visual Studio or Xcode, it is important to build the same configuration as was specified with rtime-make's `--config` option. While it is possible to build a different configuration from the IDE, selecting a different configuration does *not* update the build configuration generated for *Connext DDS Micro*.

The GUI should be started from the `RTIMEHOME` directory. If this is not the case, check that:

- The source directory is the location of `RTIMEHOME`.
- The binary directory is the location of `RTIMEBUILD`.

With the CMake GUI running:

- Press 'Configure'.
- Select a generator. You must have a compatible tool installed to process the generated files.
- Select 'Use default native compilers'.
- Press 'Done'.
- Press 'Configure'.
- Check 'Grouped'.
- Expand RTIME and select your build options. All available build options for *Connext DDS Micro* are listed here.
- Type a target name for **RTIME_TARGET_NAME**. This should be the same as the *<name>* used to create the `RTIMEBUILD` directory, that is the `RTIMEBUILD` should be on the form *<path>/<RTIME_TARGET_NAME>*.
- Press 'Configure'. All red lines should disappear. Due to how CMake works, it is strongly recommended to always press 'Configure' whenever a value is changed for a variable. Other variables may depend on the modified variable and pressing 'Configure' will mark those with a red line. No red lines means everything has been configured.
- Press 'Generate'. This creates the build-files in the `RTIMEBUILD` directory. Whenever an option is changed and Configure is re-run, press Generate again.
- Exit the GUI.

Depending on the generator, do one of the following:

- For IDE generators (such as Eclipse, Visual Studio, Xcode) open the generated solution/project files and build the project/solution.
- For command-line tools (such as make, nmake, ninja) change to the RTIMEBUILD directory and run the build-tool.

After a successful build, the output is placed in RTIMEHOME/lib/<name>.

The generated build-files may contain different sub-projects that are specific to the tool. For example, when using Xcode or Visual Studio, the following targets are available:

- ALL_BUILD - Builds all the projects.

- rti_me_<name> - Builds only the specific library. Note that that dependent libraries are built first.
- ZERO_CHECK - Runs CMake to regenerate project files in case something changed in the build input. This target does not need to be built manually.

For command-line tools, try `<tool> help` for a list of available targets to build. For example, if UNIX makefiles were generated:

```
make help
```

Creating Build Files for Connext DDS Micro Using CMake from the Command Line

Open a terminal window in the `RTIMEHOME` directory and create the `RTIMEBUILD` directory. Change to the `RTIMEBUILD` directory and invoke cmake using the following arguments:

```
cmake -G <generator> -DCMAKE_BUILD_TYPE=<Debug | Release> \
      -DCMAKE_TOOLCHAIN_FILE=<toolchain file>  \
      -DRTIME_TARGET_NAME=<target-name>
```

Depending on the generator, do one of the following:

- For IDE generators (such as Eclipse, Visual Studio, Xcode) open the generated solution/project files and build the project/solution.
- For command-line tools (such as make, nmake, ninja) run the build-tool.

After a successful build, the output is placed in *RTIMEHOME/lib/<name>*.

The generated build-files may contain different sub-projects that are specific to the tool. For example, in Xcode and Visual Studio the following targets are available:

- ALL_BUILD - Builds all the projects.
- rti_me_<name> - Builds only the specific library. Note that that dependent libraries are built first.
- ZERO_CHECK - Runs CMake to regenerate project-files in case something changed in the build input. This target does not need to be built manually.

For command-line tools, try `<tool> help` for a list of available targets to build. For example, if UNIX makefiles were generated:

```
make help
```

CMake Flags used by Connext DDS Micro

The following CMake flags (-D) are understood by *Connext DDS Micro* and may be useful when building outside of the source bundle installed by RTI. An example would be incorporating the *Connext DDS Micro* source in a project tree and invoking cmake directly on the CMakeLists.txt provided by *Connext DDS Micro*.

- `-DRTIME_TARGET_NAME=\<name\>` - The name of the target (equivalant to `--name` to rtime-make). The default value is the name of the source directory.

- `-DRTIME_CMAKE_ROOT=\<path\>` - Where to place the CMake build files. The default value is *<source>/build/cmake*.
- `-DRTIME_BUILD_ROOT=\<path\>` - Where to place the intermediate build files. The default value is *<source>/build*.
- `-DRTIME_SYSTEM_FILE=\<file\>` or an empty string - This file can be used to set the PLATFORM_LIBS variable used by *Connext DDS Micro* to link with. If an empty string is specified no system file is loaded. This option may be useful when cmake can detect all that is needed. The default value is not defined, which means try to detect the system to build for.
- `-DRTI_NO_SHARED_LIB=true` - Do not build shared libraries. The default is undefined, which means shared libraries are built. NOTE: This flag must be undefined to build shared libraries. Setting the value to false is not supported.
- `-DRTI_MANUAL_BUILDID=true` - Do not automatically generate a build ID. The default value is undefined, which means generate a new build each time the libraries are built. Setting the value to false is not supported. The build ID is in its own source and only forces a recompile of a few files. Note that it is necessary to generate a build ID at least once (this is done automatically). Also, a build ID is not supported for cmake versions less than 2.8.11 because the TIMESTAMP function does not exist.

5.2.5 Connext DDS Micro Compile Options

The *Connext DDS Micro* source supports compile-time options. These options are in general used to control:

- Enabling/Disabling features.
- Inclusion/Exclusion of debug information.
- Inclusion/Exclusion of APIs.
- Target platform definitions.
- Target compiler definitions.

NOTE: It is no longer possible to build a single library using CMake. Please refer to *Custom Build Environments* for information on customized builds.

Connext DDS Micro Debug Information

Please note that *Connext DDS Micro* debug information is independent of a debug build as defined by a compiler. In the context of *Connext DDS Micro*, debug information refers to inclusion of:

- Logging of error-codes.
- Tracing of events.
- Precondition checks (argument checking for API functions).

Unless explicitly included/excluded, the following rule is used:

- For CMAKE_BUILD_TYPE = Release, the NDEBUG preprocessor directive is defined. Defining NDEBUG includes logging, but excludes tracing and precondition checks.

- For CMAKE_BUILD_TYPE = Debug, the NDEBUG preprocessor directive is undefined. With NDEBUG undefined, logging, tracing and precondition checks are included.

To manually determine the level of debug information, the following options are available:

- **OSAPI_ENABLE_LOG** (Include/Exclude/Default)
  - Include - Include logging.
  - Exclude - Exclude logging.
  - Default - Include logging based on the default rule.
- **OSAPI_ENABLE_TRACE** (Include/Exclude/Default)
  - Include - Include tracing.
  - Exclude - Exclude tracing.
  - Default - Include tracing based on the default rule.
- **OSAPI_ENABLE_PRECONDITION** (Include/Exclude/Default)
  - Include - Include tracing.
  - Exclude - Exclude tracing.
  - Default - Include precondition checks based on the default rule.

Connext DDS Micro Platform Selection

The *Connext DDS Micro* build system looks for target platform files in *RTIMEHOME/include/osapi*. All files that match *osapi_os_*.h are listed under **RTIME_OSAPI_PLATFORM**. Thus, if a new port is added it will automatically be listed and available for selection.

The default behavior, <auto detect>, is to try to determine the target platform based on header-files. The following target platforms are known to work:

- Linux (posix)
- VOS (posix)
- QNX (posix)
- Darwin (posix)
- iOS (posix)
- Android (posix)
- Win32 (windows)
- VxWorks 6.9 and later (vxworks)

However, for custom ports this may not work. Instead the appropriate platform definition file can be selected here.

Connext DDS Micro Compiler Selection

The *Connext DDS Micro* build system looks for target compiler files in *RTIMEHOME/include/osapi*. All files that match *osapi_cc_*.h are listed under **RTIME_OSAPI_COMPILER**. Thus, if a new compiler definition file is added it will automatically be listed and available for selection.

The default behavior, <auto detect>, is to try to determine the target compiler based on header-files. The following target compilers are known to work:

- GCC (stdc)
- clang (stdc)
- MSVC (stdc)

However, for others compilers this this may not work. Instead the appropriate compiler definition file can be selected here.

Connext DDS Micro UDP Options

Checking the **RTIME_UDP_ENABLE_IPALIASES** disables filtering out IP aliases. Note that this currently only works on platforms where each IP alias has its own interface name, such as eth0:1, eth1:2, etc.

Checking the **RTIME_UDP_ENABLE_TRANSFORMS_DOC** enables UDP transformations in the UDP transport.

Checking the **RTIME_UDP_EXCLUDE_BUILTIN** excludes the UDP transport from being built.

5.2.6 Cross-Compiling Connext DDS Micro

Cross-compiling the *Connext DDS Micro* source-code uses the exact same process described in *Compiling Connext DDS Micro*, but requires a additonal *tool-chain file*. A tool-chain file is a CMake file that describes the compiler, linker, etc. needed to build the source for the target. The *Connext DDS Micro* source bundle includes a few basic, generic tool-chain files for cross-compilation. In general it is expected that users will provide their own cross-compilation tool-chain files.

To see a list of available targets, use `--list` :

```
rtime-make --list
```

By convention, RTI only provides generic tool-chain files that can be used to build for a broad range of targets. For example, the Linux target can be used to build for any Linux architecture as long as it is a self-hosted build. The same is true for Windows and Darwin systems. The VxWorks tool-chain file uses the Wind River environment variables to select the compiler.

For example, to build on a Linux machine with Kernel 2.6 and gcc 4.7.3:

```
rtime-make --target Linux --name i86Linux2.6gcc4.7.3 --config Debug --build
```

By convention, a specific name such as i86Linux2.6gcc4.4.5 is expected to only build for a specific target architecture. Note however that this cannot be enforced by the script provided by

RTI. To create a target specific tool-chain file, copy the closest matching file and add it to the *RTIMEHOME/source/Unix/resource/CMake/architectures* or *RTIMEHOME/source/windows/resource/CMake/architectures* directory.

Once a tool-chain file has been created, or a suitable file has been found, edit it as needed. Then invoke rtime-make, specifying the new tool-chain file as the target architecture. For example:

```
rtime-make --target i86Linux2.6gcc4.4.5 --config Debug --build
```

5.2.7 Custom Build Environments

The preferred method to build *Connext DDS Micro* is to use CMake. However, in some cases it may be more convenient, or even necessary, to use a custom build environment. For example:

- Embedded systems often have numerous compiler, linker and board specific options that are easier to manage in a managed build.
- The compiler cannot be invoked outside of the build environment, it may be an integral part of the development environment.
- Sometimes better optimization may be achieved if all the components of a project are built together.
- It is easier to port *Connext DDS Micro*.

Importing the Connext DDS Micro Code

The process for importing the *Connext DDS Micro* Source Code into a project varies depending on the development environment. However, in general the following steps are needed:

- Create a new project or open an existing project.
- Import the entire *Connext DDS Micro* source tree from the file-system. Note that some environments let you choose whether to make a copy only link to the original files.
- Add the following include paths:
  - <root>/include
  - <root>/src/dds_c/domain
  - <root>/src/dds_c/infrastructure
  - <root>/src/dds_c/publication
  - <root>/src/dds_c/subscription
  - <root>/src/dds_c/topic
  - <root>/src/dds_c/type
- Add a compile-time definition `-DTARGET="target name"` (note that the ” must be included).
- Add a compile-time definition `-DNDEBUG` for a release build.
- Add a compile-time definition of either `-DRTI_ENDIAN_LITTLE` for a little-endian platform or `-DRTI_ENDIAN_BIG` for a big-endian platform.

- If custom OSAPI definitions are used, add a compile-time definition -DOSAPI_OS_DEF_H="my_os_file".
- If custom compiler definitions are used, add a compile-time definition -DOSAPI_CC_DEF_H="my_cc_file.h" .

5.3 Building the Connext DDS Micro Source for FreeRTOS

5.3.1 Introduction

This user guide explains the environment used to run Micro on FreeRTOS + LwIP and is organized as follows:

- *Overview*
- *Configuration*
- *CMake Support*

5.3.2 Overview

FreeRTOS and LwIP are one of the OSs and protocol stacks where *Connext DDS Micro* is known to run. STM32F769I-DISC0 has been chosen as reference hardware. This development kit has a STM32F769NIH6 microcontroller with 2 Mbytes of Flash memory and 512 Kbytes of RAM. For a full description, please refer to the microcontroller documentation.

STM provides a toolchain called SW4STM32. SW4STM32 is a free multi-OS software environment based on Eclipse, which supports the full range of STM32 microcontrollers and associated boards. SW4STM32 includes the GCC C/C++ compiler, a GDB-based debugger, and an Eclipse based IDE.

STM also provides STM32CubeF7. STM32CubeF7 gathers all the generic embedded software components required to develop an application on the STM32F7 microcontrollers in a single package.

STM32CubeF7 also includes many examples and demonstration applications. The example *LwIP_HTTP_Server_Socket_RTOS* is particularly useful as it provides a working FreeRTOS + LwIP configuration.

The following versions of the different components have been used:

- SW4STM32 version 2.1
- STM32Cube_FW_F7 version V1.7.0
- FreeRTOS version V9.0.0
- LwIP version V2.0.0

5.3.3 Configuration

Example LwIP and FreeRTOS configurations are provided below for reference. This configuration must be tuned according to the needs. Details about how to configure these third party components can be found in the FreeRTOS and LwIP documentation.

- Example configuration for LwIP:

```
#ifndef __LWIPOPTS_H__
#define __LWIPOPTS_H__

#include <limits.h>

#define NO_SYS                  0

/* ---------- Memory options ---------- */
#define MEM_ALIGNMENT           4

#define MEM_SIZE                (50*1024)

#define MEMP_NUM_PBUF           10

#define MEMP_NUM_UDP_PCB        6

#define MEMP_NUM_TCP_PCB        10

#define MEMP_NUM_TCP_PCB_LISTEN 5

#define MEMP_NUM_TCP_SEG        8

#define MEMP_NUM_SYS_TIMEOUT    10


/* ---------- Pbuf options ---------- */
#define PBUF_POOL_SIZE          8

#define PBUF_POOL_BUFSIZE       1524

/* ---------- IPv4 options ---------- */
#define LWIP_IPV4                1

/* ---------- TCP options ---------- */
#define LWIP_TCP                1
#define TCP_TTL                 255

#define TCP_QUEUE_OOSEQ         0

#define TCP_MSS                 (1500 - 40)          /* TCP_MSS = (Ethernet MTU - IP␣
↪header size - TCP header size) */

#define TCP_SND_BUF             (4*TCP_MSS)

#define TCP_SND_QUEUELEN        (2* TCP_SND_BUF/TCP_MSS)

#define TCP_WND                 (2*TCP_MSS)


/* ---------- ICMP options ---------- */
#define LWIP_ICMP               1

```

(continues on next page)

(continued from previous page)

```
/* ---------- DHCP options ---------- */
#define LWIP_DHCP               1


/* ---------- UDP options ---------- */
#define LWIP_UDP                1
#define UDP_TTL                 255


/* ---------- Statistics options ---------- */
#define LWIP_STATS 0

/* ---------- link callback options ---------- */
#define LWIP_NETIF_LINK_CALLBACK        1

/*
   --------------------------------------
   ---------- Checksum options ----------
   --------------------------------------
*/

/*
The STM32F7xx allows computing and verifying checksums by hardware
*/
#define CHECKSUM_BY_HARDWARE


#ifdef CHECKSUM_BY_HARDWARE
  /* CHECKSUM_GEN_IP==0: Generate checksums by hardware for outgoing IP packets.*/
  #define CHECKSUM_GEN_IP                 0
  /* CHECKSUM_GEN_UDP==0: Generate checksums by hardware for outgoing UDP packets.*/
  #define CHECKSUM_GEN_UDP                0
  /* CHECKSUM_GEN_TCP==0: Generate checksums by hardware for outgoing TCP packets.*/
  #define CHECKSUM_GEN_TCP                0
  /* CHECKSUM_CHECK_IP==0: Check checksums by hardware for incoming IP packets.*/
  #define CHECKSUM_CHECK_IP               0
  /* CHECKSUM_CHECK_UDP==0: Check checksums by hardware for incoming UDP packets.*/
  #define CHECKSUM_CHECK_UDP              0
  /* CHECKSUM_CHECK_TCP==0: Check checksums by hardware for incoming TCP packets.*/
  #define CHECKSUM_CHECK_TCP              0
  /* CHECKSUM_CHECK_ICMP==0: Check checksums by hardware for incoming ICMP packets.*/
  #define CHECKSUM_GEN_ICMP               0
#else
  /* CHECKSUM_GEN_IP==1: Generate checksums in software for outgoing IP packets.*/
  #define CHECKSUM_GEN_IP                 1
  /* CHECKSUM_GEN_UDP==1: Generate checksums in software for outgoing UDP packets.*/
  #define CHECKSUM_GEN_UDP                1
  /* CHECKSUM_GEN_TCP==1: Generate checksums in software for outgoing TCP packets.*/
  #define CHECKSUM_GEN_TCP                1
  /* CHECKSUM_CHECK_IP==1: Check checksums in software for incoming IP packets.*/
```

(continues on next page)

(continued from previous page)

```
  #define CHECKSUM_CHECK_IP               1
  /* CHECKSUM_CHECK_UDP==1: Check checksums in software for incoming UDP packets.*/
  #define CHECKSUM_CHECK_UDP              1
  /* CHECKSUM_CHECK_TCP==1: Check checksums in software for incoming TCP packets.*/
  #define CHECKSUM_CHECK_TCP              1
  /* CHECKSUM_CHECK_ICMP==1: Check checksums by hardware for incoming ICMP packets.*/
  #define CHECKSUM_GEN_ICMP               1
#endif


/*
   ----------------------------------------------
   ---------- Sequential layer options ----------
   ----------------------------------------------
*/
#define LWIP_NETCONN                    1

/*
   ------------------------------------
   ---------- Socket options ----------
   ------------------------------------
*/
#define LWIP_SOCKET                     1

/*
   ---------------------------------
   ---------- OS options ----------
   ---------------------------------
*/

#define TCPIP_THREAD_NAME              "TCP/IP"
#define TCPIP_THREAD_STACKSIZE          1000
#define TCPIP_MBOX_SIZE                 6
#define DEFAULT_UDP_RECVMBOX_SIZE       2000
#define DEFAULT_TCP_RECVMBOX_SIZE       2000
#define DEFAULT_ACCEPTMBOX_SIZE         2000
#define DEFAULT_THREAD_STACKSIZE        500
#define TCPIP_THREAD_PRIO               osPriorityHigh

/**
 * LWIP_SO_RCVBUF==1: Enable SO_RCVBUF processing.
 */
#define LWIP_SO_RCVBUF                  1

#endif /* __LWIPOPTS_H__ */
```

- Example configuration for FreeRTOS:

```
#ifndef FREERTOS_CONFIG_H
```

(continues on next page)

(continued from previous page)

```
#define FREERTOS_CONFIG_H

/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

/* Ensure stdint is only used by the compiler, and not the assembler. */
#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)
 #include <stdint.h>
 extern uint32_t SystemCoreClock;
#endif

#define configUSE_PREEMPTION              1
#define configUSE_IDLE_HOOK               0
#define configUSE_TICK_HOOK               0
#define configCPU_CLOCK_HZ                (SystemCoreClock)
#define configTICK_RATE_HZ                ((TickType_t)1000)
#define configMAX_PRIORITIES              (7)
#define configMINIMAL_STACK_SIZE          ((uint16_t)128)
#define configTOTAL_HEAP_SIZE             ((size_t)(400 * 1024))
#define configMAX_TASK_NAME_LEN           (16)
#define configUSE_TRACE_FACILITY          1
#define configUSE_16_BIT_TICKS            0
#define configIDLE_SHOULD_YIELD           1
#define configUSE_MUTEXES                 1
#define configQUEUE_REGISTRY_SIZE         8
#define configCHECK_FOR_STACK_OVERFLOW    0
#define configUSE_RECURSIVE_MUTEXES       1
#define configUSE_MALLOC_FAILED_HOOK      0
#define configUSE_APPLICATION_TASK_TAG    0
#define configUSE_COUNTING_SEMAPHORES     1
#define configGENERATE_RUN_TIME_STATS     0

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES           0
#define configMAX_CO_ROUTINE_PRIORITIES (2)

/* Software timer definitions. */
#define configUSE_TIMERS             1
#define configTIMER_TASK_PRIORITY    (2)
#define configTIMER_QUEUE_LENGTH     10
#define configTIMER_TASK_STACK_DEPTH 1280

/* Set the following definitions to 1 to include the API function, or zero
```

(continues on next page)

(continued from previous page)

```
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet       1
#define INCLUDE_uxTaskPriorityGet      1
#define INCLUDE_vTaskDelete            1
#define INCLUDE_vTaskCleanUpResources  0
#define INCLUDE_vTaskSuspend           1
#define INCLUDE_vTaskDelayUntil        0
#define INCLUDE_vTaskDelay             1
#define INCLUDE_xTaskGetSchedulerState 1

/* Cortex-M specific definitions. */
#ifdef __NVIC_PRIO_BITS
 /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
 #define configPRIO_BITS         __NVIC_PRIO_BITS
#else
 #define configPRIO_BITS         4        /* 15 priority levels */
#endif

#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY   0xf

#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5

#define configKERNEL_INTERRUPT_PRIORITY   ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY <<
↪(8 - configPRIO_BITS) )

#define configMAX_SYSCALL_INTERRUPT_PRIORITY  ( configLIBRARY_MAX_SYSCALL_INTERRUPT_
↪PRIORITY << (8 - configPRIO_BITS) )

#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }

#define vPortSVCHandler    SVC_Handler
#define xPortPendSVHandler PendSV_Handler

#endif /* FREERTOS_CONFIG_H */
```

5.3.4 CMake Support

Support to compile *Connext DDS Micro* libraries for FreeRTOS using CMake has been added. It is assumed that the *Connext DDS Micro* source-bundle is downloaded and installed and that CMake is available.

1. Make sure CMake is in the path.
2. Enter the following command:

```
cd <rti_me install directory>
resource/scripts/rtime-make --target FreeRTOS --name cortexm7FreeRTOS9.
↪0gcc7.3.1 -G "Unix Makefiles" --build
```

3. The *Connext DDS Micro* libraries are available in:

```
<rti_me install directory>/lib/cortexm7FreeRTOS9.0gcc7.3.1
```

NOTE: rtime-make uses the name specified with –name to determine a few settings needed by *Connext DDS Micro*. Please refer to *Preparing for a Build* for details.

5.4 Porting RTI Connext DDS Micro

RTI Connext DDS Micro has been engineered for reasonable portability to platforms and environments which RTI does not have access to. This porting guide describes the features required by *Connext DDS Micro* to run. The target audience is developers familiar with general OS concepts, the standard C library, and embedded systems.

Connext DDS Micro uses an abstraction layer to support running on a number of platforms. The abstraction layer, OSAPI, is an abstraction of functionality typically found in one or more of the following libraries and services:

- Operating System calls
- Device drivers
- Standard C library

The OSAPI module is designed to be relatively easy to move to a new platform. All functionality, with the exception of the UDP transport which must be ported, is contained within this single module. It should be noted that although some functions may not seem relevant on a particular platform, they must still be implemented as they are used by other modules. For example, the port running on Stellaris with no OS support still needs to implement a threading model.

Please note that the OSAPI module is not designed to be a general purpose abstraction layer; its sole purpose is to support the execution of *Connext DDS Micro*.

5.4.1 Updating from Connext DDS Micro 2.4.8 and earlier

In *RTI Connext DDS Micro* 2.4.9, a few changes were made to simplify incorporating new ports. To upgrade an existing port to work with 2.4.9, follow these rules:

- Any changes to osapi_config.h should be placed in its own file (see *Directory Structure*).
- Define the OSAPI_OS_DEF_H preprocessor directive to include the file (refer to *OS and CC Definition Files*).
- For compiler-specific definitions, please refer to *OS and CC Definition Files*.
- Please refer to *Heap Porting Guide* for changes to the Heap routines that need to be ported.

5.4.2 Directory Structure

The source shipped with *Connext DDS Micro* is identical to the source developed and tested by RTI (with the exception of the the line-endings difference between the Unix and Windows source-bundles).

The source-bundle directory structure is as follows:

```
RTIMEHOME--+-- CmakeLists.txt
           |
           +-- build -- cmake --+-- Debug --+-- <ARCH> -- <project-files>
           |                    |
           |                    |
           |                    +-- Release --+-- <ARCH> -- <project-files>
           +-- doc --
           |
           +-- example
           |
           +-- include
           |
           +-- lib +-- <ARCH> -- <libraries>
           |
           +-- resource --+-- cmake
           |              |
           |              +-- scripts
           |
           +-- rtiddsgen
           |
           +-- rtiddsmag
           |
           +-- src
```

The include directory contains the external interfaces, those that are available to other modules. The src directory contains the implementation files. Please refer to *Building the Connext DDS Micro Source* for how to build the source code.

The remainder of this document focuses on the files that are needed to add a new port. The following directory structure is expected:

```
---+-- include --+-- osapi --+-- osapi_os_\<port\>.h
   |                         |
   |                         +-- osapi_cc_<compiler>.h
   |
   +-- src --+-- osapi --+-- common -- <common files>
                         |
                         +-- <port> --+-- <port>Heap.c
                                      |
                                      +-- <port>Mutex.c
                                      |
                                      +-- <port>Process.c
                                      |
                                      +-- <port>Semaphore.c
                                      |
                                      +-- <port>String.c
                                      |
                                      +-- <port>System.c
                                      |
                                      +-- <port>Thread.c
                                      |
                                      +-- <port>shmSegment.c
```

(continues on next page)

(continued from previous page)

```
                                            |
                                            +-- <port>shmMutex.c
```

The *osapi_os_<port>.h* file contains OS specific definitions for various data-types. The <port> name should be short and in lower case, for example *myos*.

The *osapi_cc_<compiler>.h* file contains compiler specific definitions. The <compiler> name should be short and in lower case, for example *mycc*. The osapi_cc_stdc.h file properly detects GCC and MSVC and it is not necessary to provide a new file if one of these compilers is used.

The <port>Heap.c, <port>Mutex.c, <port>Process.c, <port>Semaphore.c, <port>String.c and <port>System.c files shall contain the implementation of the required APIs.

NOTE: It is *not* recommended to modify source files shipped with *Connext DDS Micro*. Instead if it is desired to start with code supplied by RTI it is recommended to *copy* the corresponding sub-directory, for example posix, and rename it. This way it is easier to upgrade *Connext DDS Micro* while keeping existing ports.

5.4.3 OS and CC Definition Files

The *include/osapi/osapi_os_<port>.h* file contains OS and platform specific definitions used by OSAPI and other modules. To include the platform specific file, define **OSAPI_OS_DEF_H** as a preprocessor directive.

```
-DOSAPI_OS_DEF_H=\"osapi_os_<port>.h\"
```

It should be noted that *Connext DDS Micro* does not use auto-detection programs to detect the host and target build environment and only relies on predefined macros to determine the target environment. If *Connext DDS Micro* cannot determine the target environment, it is necessary to manually configure the correct OS definition file by defining **OSAPI_OS_DEF_H** (see above).

The *include/osapi/osapi_cc_<compiler>.h* file contains compiler specific definitions used by OSAPI and other modules. To include the platform specific file, define **OSAPI_CC_DEF_H** as a preprocessor directive.

```
-DOSAPI_CC_DEF_H=\"osapi_cc_<compiler>.h\"
```

Endianness of some platforms is determined automatically via the platform specific file, but for others either **RTI_ENDIAN_LITTLE** or **RTI_ENDIAN_BIG** must be defined manually for little-endian or big-endian, respectively.

5.4.4 Heap Porting Guide

Connext DDS Micro uses the heap to allocate memory for internal data-structures. With a few exceptions, *Connext DDS Micro* does *not* return memory to the heap. Instead, *Connext DDS Micro* uses internal pools to quickly allocate and free memory for specific types. Only the initial memory is allocated directly from the heap. The following functions must be ported:

- OSAPI_Heap_allocate_buffer
- OSAPI_Heap_free_buffer

However, if the OS and C library supports the standard malloc and free APIs define the following in the *osapi_os_<port>.h* file:

```
#define OSAPI_ENABLE_STDC_ALLOC   (1)
#define OSAPI_ENABLE_STDC_REALLOC (1)
#define OSAPI_ENABLE_STDC_FREE    (1)
```

Please refer to the OSAPI_Heap API for definition of the behavior. The available source code contains implementation in the file *osapi/<port>/<port>Heap.c*.

5.4.5 Mutex Porting Guide

Connext DDS Micro relies on mutex support to protect internal data-structures from corruption when accessed from multiple threads.

The following functions must be ported:

- OSAPI_Mutex_new
- OSAPI_Mutex_delete
- OSAPI_Mutex_take_os
- OSAPI_Mutex_give_os

Please refer to the OSAPI_Mutex API for definition of the behavior. The available source code contains implementation in the file *osapi/<port>/<port>Mutex.c*

5.4.6 Semaphore Porting Guide

Connext DDS Micro relies on semaphore support for thread control. If *Connext DDS Micro* is running on a non pre-emptive operating system with no support for IPC and thread synchronization, it is possible to implement these functions as no-ops. Please refer to *Thread Porting Guide* for details regarding threading.

The following functions must be ported:

- OSAPI_Semaphore_new
- OSAPI_Semaphore_delete
- OSAPI_Semaphore_take
- OSAPI_Semaphore_give

Please refer to the OSAPI_Semaphore API for definition of the behavior. The available source code contains implementation in the file *osapi/<port>/<port>Semaphore.c*.

5.4.7 Process Porting Guide

Connext DDS Micro only uses the process API to retrieve a unique ID for the applications.

The following functions must be ported:

- OSAPI_Process_getpid

Please refer to the OSAPI_Process_getpid API for definition of the behavior. The available source code contains implementation in the file *osapi/<port>/<port>Process.c*.

5.4.8 System Porting Guide

The system API consists of functions which are more related to the hardware on which *Connext DDS Micro* is running than on the operating system. As of *Connext DDS Micro* 2.3.1, the system API is implemented as an interface as opposed to the previous pure function implementation. This change makes it easier to adapt *Connext DDS Micro* to different hardware platforms without having to write a new port.

The system interface is defined in OSAPI_SystemI, and a port must implement all the methods in this structure. In addition, the function OSAPI_System_get_native_interface must be implemented. This function must return the system interface for the port (called the native system interface).

The semantics for the methods in the interface are exactly as defined by the corresponding system function. For example, the method OSAPI_SystemI::get_time must behave exactly as that described by OSAPI_System_get_time.

The following system interface methods must be implemented in the OSAPI_SystemI structure:

- OSAPI_SystemI::get_timer_resolution
- OSAPI_SystemI::get_time
- OSAPI_SystemI::start_timer
- OSAPI_SystemI::stop_timer
- OSAPI_SystemI::generate_uuid
- OSAPI_SystemI::get_hostname
- OSAPI_SystemI::initialize
- OSAPI_SystemI::finalize

Please refer to the OSAPI_System API for definition of the behavior. The available source code contains implementation in the file: *osapi/<port>/<port>System.c*.

Migrating a 2.2.x port to 2.3.x

In *Connext DDS Micro* 2.3.x, changes where made to how the system API is implemented. Because of these changes, existing ports must be updated, and this section describes how to make a *Connext DDS Micro* 2.2.x port compatible with *Connext DDS Micro* 2.3.x.

If you have ported *Connext DDS Micro* 2.2.x the following steps will make it compatible with version 2.3.x:

- Rename the following functions and make them private to your source code. For example, rename OSAPI_System_get_time to OSAPI_MyPortSystem_get_time etc.
  - OSAPI_System_get_time
  - OSAPI_System_get_timer_resolution

   - OSAPI_System_start_timer
   - OSAPI_System_stop_timer
   - OSAPI_System_generate_uuid
- Implement the following new methods.
   - OSAPI_SystemI::get_hostname
   - OSAPI_SystemI::initialize
   - OSAPI_SystemI::finalize
- Create a system structure for your port using the following template:

```
struct OSAPI_MyPortSystem
{
    struct OSAPI_System _parent;

    Your system variable
};

static struct OSAPI_MyPortSystem OSAPI_System_g;

/* OSAPI_System_gv_system is a global system variable used by the
 * generic system API. Thus, the name must be exactly as
 * shown here.
 */
struct OSAPI_System * OSAPI_System_gv_system = &OSAPI_System_g._parent;
```

- Implement OSAPI_System_get_native_interface method and fill the OSAPI_SystemI structure with all the system methods.

5.4.9 Thread Porting Guide

The thread API is used by *Connext DDS Micro* to create threads. Currently only the UDP transport uses threads and it is a goal to keep the generic *Connext DDS Micro* core library free of threads. Thus, if *Connext DDS Micro* is ported to an environment with no thread support, the thread API can be stubbed out. However, note that the UDP transport must be ported accordingly in this case; that is, all thread code must be removed and replaced with code appropriate for the environment.

The following functions must be ported:

- OSAPI_Thread_create
- OSAPI_Thread_sleep

Please refer to the OSAPI_Thread API for definition of the behavior. The available source code contains implementation in the file *srcC/osapi/<platform>/Thread.c*.

Chapter 6

Working with RTI Connext DDS Micro and RTI Connext DDS

In some cases, it may be necessary to write an application that is compiled against both *RTI Connext DDS Micro* and *RTI Connext DDS*. In general this is not easy to do because *RTI Connext DDS Micro* supports a very limited set of features compared to *RTI Connext DDS*.

However, due to the nature of the DDS API and the philosophy of declaring behavior through QoS profiles instead of using different APIs, it may be possible to share common code. In particular, *RTI Connext DDS* supports configuration through QoS profile files, which eases the job of writing portable code.

Please refer to *Introduction* for an overview of features and what is supported by *RTI Connext DDS Micro*. Note that *RTI Connext DDS* supports many extended APIs that are not covered by the DDS specification, for example APIs that create DDS entities based on QoS profiles.

6.1 Development Environment

There are no conflicts between *RTI Connext DDS Micro* and *RTI Connext DDS* with respect to library names, header files, etc. It is advisable to keep the two installations separate, which is the normal case.

RTI Connext DDS Micro uses the environment variable RTIMEHOME to locate the root of the *RTI Connext DDS Micro* installation.

RTI Connext DDS uses the environment variable NDDSHOME to locate the root of the *RTI Connext DDS* installation.

6.2 Non-standard APIs

The DDS specification omits many APIs and policies necessary to configure a DDS application, such as transport, discovery, memory, logging, etc. In general, *RTI Connext DDS Micro* and *RTI Connext DDS* do not share APIs for these functions.

It is recommended to configure *RTI Connext DDS* using QoS profiles as much as possible.

6.3 QoS Policies

QoS policies defined by the DDS standard behave the same between *RTI Connext DDS Micro* and *RTI Connext DDS*. However, note that *RTI Connext DDS Micro* does not always support all the values for a policy and in particular unlimited resources are not supported.

Unsupported QoS policies are the most likely reason for not being able to switch between *RTI Connext DDS Micro* and *RTI Connext DDS*.

6.4 Standard APIs

APIs that are defined by the standard behave the same between *RTI Connext DDS Micro* and *RTI Connext DDS*.

6.5 IDL Files

RTI Connext DDS Micro and *RTI Connext DDS* use the same IDL compiler (rtiddsgen) and *RTI Connext DDS Micro* typically ships with the latest version. However, *RTI Connext DDS Micro* and *RTI Connext DDS* use different templates to generate code and it is not possible to share the generated code. Thus, while the same IDL can be used, the generated output must be saved in different locations.

@section microcore_interop Interoperability

In general, *RTI Connext DDS Micro* and *RTI Connext DDS* are wire interoperable, unless noted otherwise.

All RTI products, aside from *RTI Connext DDS Micro*, are based on *RTI Connext DDS*. Thus, in general *RTI Connext DDS Micro* is compatible with RTI tools and other products. The following sections provide additional information for each product.

When trying to establish communication between an *RTI Connext DDS Micro* application that uses the Dynamic Participant / Static Endpoint (DPSE) discovery module and an RTI product based on *RTI Connext DDS*, every participant in the DDS system must be configured with a unique participant name. While the static discovery functionality provided by *RTI Connext DDS* allows participants on different hosts to share the same name, *RTI Connext DDS Micro* requires every participant to have a different name to help keep the complexity of its implementation suitable for smaller targets.

6.6 Admin Console

Admin Console can discover and display *RTI Connext DDS Micro* applications that use full dynamic discovery (DPDE). When using static discovery (DPSE), it is required to use the Limited Bandwidth Endpoint Discovery (LBED) that is available as a separate product for *RTI Connext DDS*. With the library a configuration file with the discovery configuration must be provided (just as in the case for products such as Routing Service, etc.). This is provided through the QoS XML file.

Data can be visualized from *RTI Connext DDS Micro* DataWriters. Keep in mind that *RTI Connext DDS Micro* does not currently distribute type information and the type information has to be provided through an XML file using the "Create Subscription" dialog. Unlike some other products, this information cannot be provided through the QoS XML file. To provide the data types to Admin Console, first run the code generator with the `-convertToXml` option:

```
rtiddsgen -convertToXml <file>
```

Then click on the "Load Data Types from XML file" hyperlink in the "Create Subscription" dialog and add the generated IDL file.

Other Features Supported:

- Match analysis is supported.
- Discovery-based QoS are shown.

The following resource-limits in *RTI Connext DDS Micro* must be incremented as follows when using Admin Console:

- Add 24 to DDS_DomainParticipantResourceLimitsQosPolicy::remote_reader_allocation
- Add 24 to DDS_DomainParticipantResourceLimitsQosPolicy::remote_writer_allocation
- Add 1 to DDS_DomainParticipantResourceLimitsQosPolic::remote_participant_allocation
- Add 1 to DDS_DomainParticipantResourceLimitsQosPolicy::remote_participant_allocation if data-visualization is used

RTI Connext DDS Micro does not currently support any administration capabilities or services, and does not match with the Admin Console DataReaders and DataWriters. However, if matching DataReaders and DataWriters are created, e.g., by the application, the following resource must be updated:

- Add 48 to DDS_DomainParticipantResourceLimitsQosPolicy::matching_writer_reader_pair_allocation

6.7 Distributed Logger

This product is not supported by *RTI Connext DDS Micro*.

6.8 LabVIEW

The LabVIEW toolkit uses *RTI Connext DDS*, and it must be configured as any other *RTI Connext DDS* application. A possible option is to use the builtin *RTI Connext DDS* profile: BuiltinQosLib::Generic.ConnextMicroCompatibility.

6.9 Monitor

This product is not supported by *RTI Connext DDS Micro*.

6.10 Recording Service

6.10.1 RTI Recorder

RTI Recorder is compatible with *RTI Connext DDS Micro* in the following ways:

- If static endpoint discovery is used, Recorder is compatible starting with version 5.1.0.3 and onwards.
- If dynamic endpoint discovery is used, Recorder is compatible with *RTI Connext DDS Micro* the same way it is with any other DDS application.
- In both cases, type information has to be provided via XML. Read Recording Data with *RTI Connext DDS Micro* for more information.

6.10.2 RTI Replay

RTI Replay is compatible with *RTI Connext DDS Micro* in the following ways:

- If static endpoint discovery is used, Replay is compatible starting with version 5.1.0.3 and onwards.
- If dynamic endpoint discovery is used, Replay is compatible with *RTI Connext DDS Micro* the same way it is with any other DDS application.
- In both cases, type information has to be provided via XML. Read Recording Data with *RTI Connext DDS Micro* for more information on how to convert from IDL to XML.

6.10.3 RTI Converter

Databases recorded with *RTI Connext DDS Micro* contains no type information in the DCPSPublication table, but the type information can be provided via XML. Read Recording Data with *RTI Connext DDS Micro* for more information on how to convert from IDL to XML.

6.11 Spreadsheet Addin

RTI Connext DDS Micro can be used with Spreadsheet Add-in starting with version 5.2.0. The type information must be loaded from XML files.

6.12 Wireshark

Wireshark fully supports *RTI Connext DDS Micro*.

6.13 Persistence Service

RTI Connext DDS Micro only supports VOLATILE and TRANSIENT_LOCAL durability and does not support the use of Persistence Service.

Chapter 7

API Reference

RTI Connext DDS Micro features API support for C and C++. Select the appropriate language below in order to access the corresponding API Reference HTML documentation.

- C API Reference
- C++ API Reference

Chapter 8

Release Notes

8.1 Supported Platforms and Programming Languages

Connext DDS Micro supports the C and traditional C++ language bindings.

Note that RTI only tests on a subset of the possible combinations of OSs and CPUs. Please refer to the following table for a list of specific platforms and the specific configurations that are tested by RTI.

OS	CPU	Compiler	RTI Architecture Abbreviation
Red Hat® Enterprise Linux® 6.0, 6.1 (Kernel version 2.6)	x86	gcc 4.4.5	i86Linux2.6gcc4.4.5
Ubuntu® 18.04 (Kernel version 4)	x64	gcc 7.3.0	x64Linux4gcc7.3.0
Ubuntu 16.04 (Kernel version 3)	x86	gcc 5.4.0	i86Linux3gcc5.4.0
PPC Linux (Kernel version 2.6)	ppc7400	gcc 3.3.3	ppc7400Linux2.6gcc3.3.3
macOS® 10.16	x64	clang 8.0	x64Darwin16clang8.0
QNX® 7.0	armv8	qcc 5.4.0	armv8QNX7.0.0qcc_gpp5.4.0
QNX 6.6	armv7a	qcc 4.7.3	armv7aQNX6.6.0qcc_cpp4.7.3
QNX 6.6	i86	qcc 4.7.3	i86QNX6.6qcc_cpp4.7.3
Windows® 7	x86	Visual Studio® 2010	i86Win32VS2010
Windows® 7	x64	Visual Studio® 2015	x64Win64VS2015

8.2 Compatibility

For backward compatibility information between 3.0.2 and previous releases, see the *Migration Guide* on the RTI Community Portal (https://community.rti.com/documentation).

8.3 What's New in 2.4.12

8.3.1 Shared UDP port for discovery and user-data in a DomainParticipant

This release allows sharing a UDP port per DomainParticipant for discovery and user-traffic. The advantage is that *Connext DDS Micro* will create a single receive thread for unicast instead of two.

The disadvantage is that this port mapping is not compliant with the DDS Interoperability Wire Protocol and communication with other DDS implementations might not be possible.

This feature may only be used if multicast *or* unicast is used for both discovery and user traffic. If both unicast *and* multicast are enabled this feature cannot be used.

To enable this feature assign the same value to both builtin and user port offsets in RtpsWellKnownPorts_t.

8.3.2 DomainParticipants no longer allocate dynamic memory during deletion

DomainParticipants will no longer allocate dynamic memory during deletion.

8.3.3 New QoS parameter to set maximum outstanding samples allowed for remote DataWriter

A new QoS parameter has been exposed for the endpoint discovery endpoints in the dynamic endpoint discovery plugin (DPDE). The new field, *max_samples_per_remote_builtin_endpoint_writer* in DPDE_DiscoveryPluginProperty, can be set to increase the number of samples a remote writer may have per builtin endpoint reader and thus decrease network traffic. Please refer to the DPDE for a description of this new parameter.

8.3.4 New QoS parameter to adjust preemptive ACKNACK period

A new QoS parameter has been introduced to expose the preemptive ACKNACK period on DataReaders. The new parameter is configured with:

- *DDS_DataReaderQos.protocol.rtps_reliable_reader.nack_period* for user data readers
- *builtin_endpoint_reader_nack_period* for the builtin discovery endpoints in the Dynamic discovery plugin

Please refer to *API Reference* API for details.

8.3.5 Deserialization of Presentation QoS policy

This release provides better support for the Presentation QoS policy. Previously this QoS policy was not supported by the DataWriter; the default value was assumed for a discovered DataReader, which caused an "Unknown QoS" warning when it was received. In this release, DataWriters will deserialize the Presentation QoS policy and check for compatibility.

8.4 What's Fixed in 2.4.12

8.4.1 Examples used DomainParticipant_register_type instead of FooTypeSupport_register_type

In previous versions the examples registered types using "DDS_DomainParticipant_register_type()" instead of the recommended "FooTypeSupport_register_type()". This version has updated the examples to use the recommended "FooTypeSupport_register_type()" instead.

[RTI Issue ID MICRO-1922]

8.4.2 A DataReader and DataWriter with incompatible liveliness kind and infinite lease_duration matched

In previous versions *Connext DDS Micro* allowed a DataWriter to match a DataReader if the liveliness kind was incompatible *but* the liveliness duration was infinite. However, the OMG DDS specification mandates stricter matching rules and in this version a DataReader and DataWriter will *only* match when both the liveliness duration and kind are compatible:

1. Requested Liveliness Lease duration is greater than or equal to the Offered lease duration.

2. Requested Liveliness kind is less than or equal to the Offered Liveliness kind where AUTOMATIC_LIVELINESS_KIND < MANUAL_BY_PARTICIPANT_LIVELINESS_KIND < MANUAL_BY_TOPIC_LIVELINESS_KIND.

Note that this did not affect communication between *Connext DDS Micro* applications since with an infinite liveliness duration, the liveliness will never expire, regardless of kind.

[RTI Issue ID MICRO-2007]

8.4.3 Warning at compilation time for FreeRTOS port

An incompatible pointer type warning was printed at compilation time when compiling for FreeRTOS. This issue has been resolved.

[RTI Issue ID MICRO-2090]

8.4.4 Using *DDS_NOT_ALIVE_INSTANCE_STATE* caused compilation error in C and C++

Using the constant DDS_NOT_ALIVE_INSTANCE_STATE caused a linker error due to a missing definition. This issue has been resolved.

[RTI Issue ID MICRO-2243]

8.4.5 *Seq_copy()* did not work when the source sequence is a loaned/discontiguous sequence

Calling *FooSeq_copy()* on a loaned or discontiguous sequence did not work correctly. This issue has been fixed.

[RTI Issue ID MICRO-2053]

8.4.6 Warnings when compiling the example generated by Code Generator

When compiling the example generated by rtiddsgen, the compiler may have given warnings about unused variables. The generated code has been updated to avoid these warnings.

[RTI Issue ID MICRO-1700]

8.4.7 Unable to generate code for XML or XSD defined types

Previous releases of *Connext DDS Micro* did not include the XML and XSD schemas required to generate type-support code from XML or XSD files. This issue has been resolved.

[RTI Issue ID MICRO-1709]

8.4.8 Linker error in C++ application when C types were used

Compiling generated C type-support code as C++ caused compilation errors. This issue has been resolved.

[RTI Issue ID MICRO-1750]

8.4.9 Failure to link for VxWorks RTP using shared libraries compiled with CMake

Due to use of incorrect compiler and linker options for VxWorks RTP mode a linker error occurred when compiling projects generated with CMake®. This issue has been resolved.

[RTI Issue ID MICRO-1909]

8.4.10 rtiddsgen may have failed on Windows systems when -jre was specified

The rtiddsen -jre option did not accept paths with spaces. This issue has been resolved.

[RTI Issue ID MICRO-1952]

8.4.11 rtime-make did not work when it was started from different shell than Bash

rtime-make requires Bash on Unix systems. However it did not explicitly launch Bash and would fail if started from a Bash incompatible shell. This has been fixed.

[RTI Issue ID MICRO-2013]

8.4.12 Linker error when using shared libraries on VxWorks systems

There was a linker error when compiling the examples for ppc604Vx6.9gcc4.3.3 using shared libraries. The compiler reported that the libraries could not be found. This issue has been resolved.

[RTI Issue ID MICRO-1841]

8.4.13 A run-time error may have occurred on Windows or when compiling for FACE when using hostnames in the peer list

Due to incorrect use of the getaddrinfo() API on Windows or POSIX when compiling for FACE, a run-time error may have occured when resolving hostnames. This issue has been fixed.

[RTI Issue ID MICRO-1957]

8.4.14 Entity ID generation was not thread-safe

Entity ID generation for DataReaders and DataWriters was not thread-safe and may have lead to duplicate entity IDs. This problem has been resolved.

[RTI Issue ID MICRO-2104]

8.4.15 DomainParticipant creation failed if active interface had invalid IP

An active interface without a valid IP address assigned may have caused DomainParticipant creation to fail. This problem has been resolved. Now if an interface with an invalid IP address is used, it will be ignored and the DomainParticipant will still be created.

[RTI Issue ID MICRO-1602]

8.4.16 rtime-make did not work when there was a space in the installation path

The rtime-make script did not work when *Connext DDS Micro* was installed in a directory path containing spaces. This issues has been resolved.

[RTI Issue ID MICRO-1622]

8.4.17 Sample filtering methods were always added to the subscriber code for C

The generated subscriber example code always included code to filter sample-based fields in the IDL type. However, if the generated IDL file was modified to exclude these fields, the code would fail to compile. The generated code now includes instructions for how to filter instead.

[RTI Issue ID MICRO-1980]

8.4.18 'Failure to give mutex' error

In Connext DDS Micro 2.4.11, a subtle race condition may have occurred on multi-core machines. When this happened, an error message about failing to give a mutex would be printed: error code (EC) 44 in module 1 (OSAPI). This problem has been resolved.

[RTI Issue ID MICRO-2095]

8.4.19 UDP interface warning using valid interfaces

Connext DDS Micro logged a warning if no new interfaces were added for each address listed in enabled_transports. This applied to the enabled_transports field in the DiscoveryQosPolicy and UserTrafficQosPolicy in the DomainParticipantQos, and the DDS_TransportQosPolicy in the DataReaderQos and DataWriterQos. This problem has been resolved. Now Connext DDS Micro will only log a warning if no new interfaces are added per enabled transport.

[RTI Issue ID MICRO-2018]

8.4.20 A DataReader May Stop Receiving Samples When Filtering Callbacks Are Used

When using *on_before_deserialize()* or *on_before_commit()* to drop samples the DataReader may have been depleted of resources and stop receiving data. This issue has been fixed.

[RTI Issue ID MICRO-1930]

8.4.21 DDS_WaitSet_wait() returned DDS_RETCODE_ERROR if unblocked with no active conditions

An application that used a combination of polling a DataReader and blocking on a DDS_WaitSet may have caused *DDS_WaitSet_wait()* to return DDS_RETCODE_ERROR. This happened if the DDS_WaitSet was unblocked by an attached condition, but there were no active conditions. This problem has been resolved.

[RTI Issue ID MICRO-2115]

8.4.22 Large timeout values may have caused segmentation fault

Timeout values larger than 2000s may have caused a segmentation fault during creation of DDS entities. This issue has been fixed.

[RTI Issue ID MICRO-2192]

8.4.23 HelloWorld_dpde_waitset C++ example uses wrong loop variable for printing data

When multiple samples are loaned by calling take, the HelloWorld_dpde_waitset C++ example uses the wrong loop variable, *i*, with *data_seq* instead of the correct index *b*. This issue has been resolved.

[RTI Issue ID MICRO-2158]

8.4.24 WaitSet_wait returned generic error when returned condition sequence exceeded capacity

If the number of returned conditions exceeded the maximum size of the returned condition sequence, a generic error, DDS_RETCODE_ERROR, was returned instead of the expected error, DDS_RETCODE_OUT_OF_RESOURCES. This problem has been resolved.

[RTI Issue ID MICRO-1933]

8.4.25 Publication handle not set in SampleInfo structure when on_before_sample_commit() called

The *publication_handle* member of the DDS_SampleInfo structure passed to a DataReader's *on_before_sample_commit()* function was not set. This issue has been fixed.

[RTI Issue ID MICRO-2121]

8.4.26 Duplicate DATA messages are sent to multicast in some cases

Duplicate DATA messages were sent to multicast when multiple DataReaders were configured with multicast and unicast receive addresses. This issue has been fixed.

[RTI Issue ID MICRO-2043]

8.4.27 GUID generation on QNX for processes run one after another may lead to duplicate GUIDs

On QNX systems, two processes run one after another in quick order may end up with the same GUID. The probability of GUID reuse has been reduced in this release.

[RTI Issue ID MICRO-2109]

8.4.28 Read/take APIs returned more than *depth* samples if an instance returned to alive without application reading *NOT_ALIVE* sample

If an instance transitioned from *NOT_ALIVE_NO_WRITERS* or *NOT_ALIVE_DISPOSED* to *ALIVE* and the application did not read/take the sample indicating *NOT_ALIVE_NO_WRIT-*

ERS or *NOT_ALIVE_DISPOSED*, the number of samples returned would exceed the *depth* set by the History QoS policy. This issue has been fixed.

[RTI Issue ID MICRO-2196]

8.4.29 Segmentation fault if *OSAPI_Semaphore_give()* was called from one thread while another called *OSAPI_Semaphore_delete()*

An application may have terminated with a segmentation fault if *OSAPI_Semaphore_give()* was called from one thread while another called *OSAPI_Semaphore_delete()* on Unix-like systems. This issue has been resolved.

[RTI Issue ID MICRO-2209]

8.4.30 Communication problems between Connext DDS Professional 6 and Connext DDS Micro 2.4.11

Connext DDS Professional 6 advertises support for RTPS protocol version 2.3, while Connext DDS Micro 2.4.11 and earlier only accepted RTPS 2.1. Therefore tools such as Admin Console 6.0.0 did not properly discover entities from a Micro 2.4.11 application. This release of Connext DDS Micro complies with RTPS 2.1 and later minor versions (such as 2.3). Unsupported RTPS messages are ignored.

[RTI Issue Id MICRO-2008]

8.4.31 OSAPI_System_get_ticktime() not implemented for FreeRTOS

OSAPI_System_get_ticktime() was not implemented for FreeRTOS. An application using a finite DDS deadline or liveliness would have a run-time failure. This issue has been resolved.

[RTI Issue ID MICRO-2240]

8.5 Previous Releases

8.5.1 What's New in 2.4.11

Support for ThreadX/NetX

Support for the ThreadX operating system, version 5.7, and the NetX TCP/IP network stack, version 5.9.

Batching (reception only)

Batching reception. Please refer to the new user's manual UserManuals_Batching for details.

UDP Transformations

Please refer to the new user's manual ref UserManuals_UDPTransform for details.

Optionally exclude builtin UDP Transport from compilation

Setting the flag -DRTIME_UDP_EXCLUDE_BUILTIN=1 excludes the UDP transport from being built. This setting can be useful if communication is done using only shared memory, INTRA, or a custom UDP transport.

Publication handle of DataWriter now provided upon DataReaderListener sample loss

When the DDS_DataReaderListener's on_sample_lost event is triggered, the returned DDS_SampleLostStatus.sample_info now contains the publication_handle of the DataWriter that originally wrote the lost sample(s).

DataWriters offer TOPIC presentation

Connext DDS Micro DataWriters now offer the DDS_TOPIC_PRESENTATION_QOS presentation (when coherent_access = FALSE). This presentation is compatible with any reader using DDS_TOPIC_PRESENTATION_QOS and DDS_INSTANCE_PRESENTATION_QOS, when ordered_access = TRUE and ordered_access = FALSE.

Micro readers will remain unchanged and will only support DDS_INSTANCE_PRESENTATION_QOS when ordered_access = FALSE.

New warning if a configured UDP transport does not have any interface

A warning in logs has been added to notify you when a configured UDP transport does not have any interface. This condition normally indicates a wrong UDP configuration, which might result in discovery and/or communication failure.

8.5.2 What's Fixed in 2.4.11

MICRO-1814 Incorrect thread ID returned for VxWorks RTP

The function OSAPI_Thread_self() when called by a VxWorks Real-Time Process (RTP) always returned the (process) ID of the RTP, even for tasks spawned by the RTP. This issue has been fixed.

[RTI Issue ID MICRO-1814]

NULL listener and non-empty status mask not allowed for C++ DataReader

A C++ DataReader was incorrectly not allowed to be created with a NULL DataReaderListener and a non-empty status mask (i.e., not DDS_STATUS_MASK_NONE).

[RTI Issue ID MICRO-1807]

accept_unknown_peers did not work when Shared Memory transport was enabled in RTI Connext DDS Pro

When *Connext DDS Micro* discovered a RTI Connext DDS Pro application with Shared Memory transport enabled, *Connext DDS Micro* failed to correctly use the UDPv4 locators instead.

This issue has been fixed.

[RTI Issue ID MICRO-1798]

Calling FooSeq_set_maximum() repeatedly with the same maximum size results in seg-fault

In RTI *Connext DDS Micro* 2.4.10.x and earlier, calling FooSeq_set_maximum() repeatedly with the same maximum size on an IDL sequence type containing non-primitive types (such as enums or other structures) caused a segmentation fault.

This issue has been fixed.

[RTI Issue ID MICRO-1786]

CMake reports error if CMake version 2.8.10.2 or 2.8.10.1 is used

Connext DDS Micro buildable sources can not be compiled with CMake versions 2.8.10.1 or 2.8.10.2.

This issue has been fixed.

[RTI Issue ID MICRO-1748]

OS error code (errno) not logged if sendto() returned error

The OS error code (errno) was not correctly logged if sendto() returned an error.

This issue has been fixed.

[RTI Issue ID MICRO-1712]

Codegen might generate an incorrect pub/sub example if opction "-create typefiles" is not used

Wrong example code is generated in case rtiddsgen is executed with option -create examplefiles and option -create typefiles is NOT used.

This issue has been fixed.

[RTI Issue ID MICRO-1696]

Generated examples use always the last structure in the idl

Examples generated using Codegen use always the last structure in the idl file, even if it is not top-level.

This issue has been fixed.

[RTI Issue ID MICRO-1694]

Instance might not have been disposed or unregistered under some conditions

Unregistered or disposed samples were not processed when preceded by a GAP sub-message within the same RTPS message.

This issue has been fixed.

[RTI Issue ID MICRO-1692]

Reliable Endpoints with only multicast locators may not communicate

A reliable DataReader configured with only multicast (no unicast) locator(s) may have failed to discover or communicate with a reliable DataWriter. Both built-in discovery endpoints and user-data endpoints were affected.

This issue has been fixed.

[RTI Issue ID MICRO-1687]

Access to DDSEntity instance handles from C++ API

Users of RTI Connext DDS Micro's C++ API can now access instance handles of any DDS entity using method DDSEntity::get_instance_handle.

[RTI Issue ID MICRO-1681]

Syntax changed for initial peer participant index range

When configuring the initial peers of a DomainParticipant (e.g. DDS_DomainParticipantQos.discovery.initial_peers), the syntax for specifying a range of participant indices for a peer locator has changed: a hyphen is now the separator, replacing a comma. In general, a peer "[x-y]@<address>" means that participant discovery messages will be sent to the address for participant indices x through y.

[RTI Issue ID MICRO-1680]

lookup_instance() is not thread safe

The lookup_instance() was not thread safe in *Connext DDS Micro* 2.4.10.x and earlier. If an application was calling lookup_instance() from both a listener and a WaitSet/polling thread at the same time, the instance handle could be corrupted.

This issue has been fixed.

[RTI Issue ID MICRO-1679]

CMakeLists.txt and README.txt created when they should not

Codegen generates project files CMakeLists.txt and README.txt are generated even when project files are not generated.

This issue has been fixed.

[RTI Issue ID MICRO-1673]

No communication when DomainParticipant used same GUID as another DomainParticipant in different domain

Given an application that creates DomainParticipants in different DDS domains, a DomainParticipant created with the same Participant GUID (i.e., the GUID Prefix portion of the GUID) as created for a DomainParticipant in a different domain will fail to discover or communicate with other endpoints within its own domain. A workaround would be for the application to assign unique GUIDs for all DomainParticipants across all domains. This issue has been fixed.

This issue has been fixed.

[RTI Issue ID MICRO-1671]

Compiler error might happen when lwIP is used

An incorrectly defined compiler macro causes a compilation error when lwIP stack is used and LWIP_DNS is defined.

This issue has been fixed.

[RTI Issue ID MICRO-1664]

Wrong C++ code generated for unkeyed types when using IDL modules and -namespace option

Code generated with the following command failed if a struct with the same name was defined in two namespaces, and the first namespace did not have any key:

rtiddsgen -micro -example HelloWorld.idl -replace -language C++ -namespace

This issue has been fixed.

[RTI Issue ID MICRO-1663]

DDS_WaitSet_wait does not work if OSAPI_Semaphore_take() returns an error

DDS_WaitSet_wait does not work if OSAPI_Semaphore_take() returns an error; RETCODE_PRECONDITION_NOT_MET is always returned.

This issue has been fixed.

[RTI Issue ID MICRO-1658]

Log buffer could overflow on 64-bit architectures, causing application crash

The log buffer may have overflowed on 64-bit architectures and caused an application crash.

This issue has been fixed.

[RTI Issue ID MICRO-1657]

Fix API realloc in Windows OSAPI

Windows implementation of function realloc did not allow a NULL input pointer, this is wrong and posix implementation and Windows API allow it. This has the effect that function DDS_String_replace() fails when the input string is a NULL pointer.

This issue has been fixed.

[RTI Issue ID MICRO-1655]

New samples for an instance may not be received if an instance goes back to ALIVE when using read()

Due to an issue in the resource calculation for the DataReader, new samples for an instance may not have been received if the instance went back to ALIVE when using any of the read() APIs.

This issue has been fixed.

[RTI Issue ID MICRO-1651]

INTRA transport caused subscription matches to use additional resources

An issue in the matching between a reader and writer caused a reader to be matched with the same writer twice if auto enable was set to FALSE.

This issue has been fixed.

[RTI Issue ID MICRO-1650]

Resolved memory leak in class RTRegistry

When using previous versions of *Connext DDS Micro*, C++ applications might have experienced resource leakage upon finalization of middleware resources using the method DDSDomainParticipantFactory::finalize_instance. The leaks were caused by unfreed memory blocks still owned by the class RTRegistry, and they have now been resolved. No additional action is required of users.

This issue has been fixed.

[RTI Issue ID MICRO-1637]

Windows Debug DLLs are built without debug information

Windows Debug DLLs are built without debug information what prevents debugging. This is happening when building with CMake or the rtime-make script.

This issue has been fixed.

[RTI Issue ID MICRO-1634]

Use hardcoded build ID when not compiling with CMake

When compiling using CMake or the script rtime-make, *Connext DDS Micro* libraries have a build lD (buildid), which consist of the current time and date. A hardcoded constant ID is used as the build ID when compilation is not done using CMake or the script rtime-make.

This issue has been fixed.

[RTI Issue ID MICRO-1632]

Example makefiles do not support 64bit compilation

Example makefiles used always option -m32. This has been changed to use -m32 or -m64 depending on the platform configuration.

Examples can be compiled now for 32 and 64 bits platforms.

This issue has been fixed.

[RTI Issue ID MICRO-1628]

Compilation error might happen when code is generated using option -namespace

Compilation error fixed in generated source code when option -namespace is used and IDL file has modules and compilation uses shared libraries.

This issue has been fixed.

[RTI Issue ID MICRO-1620]

8.5.3 What's New in 2.4.10.4

Batching (reception only)

This release includes batching reception. Please refer to the new user manual for Batching for details.

C++ examples

A new C++ example using Waitsets (HelloWorld_dpde_waitset) is included.

8.5.4 What's Fixed in 2.4.10.4

Improve KEEP_LAST

To reclaim resources in version 2.4.10 and earlier the DataReader cache tries to remove the oldest sample only. If that is on loan it cannot be removed and in case a new sample is received it cannot be added to the DataReader cache.

This issue has been fixed.

[RTI Issue ID MICRO-1754]

Locator might be duplicated when NAT is configured

When Network Address Translation (NAT) is configured in the transport UDP properties, a duplicated locator might be sent in discovery packets.

This issue has been fixed.

[RTI Issue ID MICRO-1756]

Segmentation fault might happen when a DataReader cannot be created

If the creation of a DataReader fails before all fields in the DataReader structure are initialized, a NULL pointer access may have occur while finalizing the already created objects.

This issue has been fixed.

[RTI Issue ID MICRO-1755]

CMake reports error if CMake version 2.8.10.2 or 2.8.10.1 is used

RTI Connext DDS Micro buildable sources could not be compiled with CMake 2.8.10.1 or 2.8.10.2.

This issue has been fixed.

[RTI Issue ID MICRO-1748]

Wrong TUDP locator kind sent when using UDP transformations

When using UDP transformations the locator kind was always set as 0, intead of the configured value in ref UDP_InterfaceFactoryProperty.transform_locator_kind

This issue has been fixed.

[RTI Issue ID MICRO-1685]

Compile shipped examples for a 64 bits architecture

Before this release shipped examples makefiles could only compile 32 bits architectures. Makefiles have been modified to support also 64 bits architectures.

This issue has been fixed.

[RTI Issue ID MICRO-1628]

OSAPI_Heap_realloc() Windows implementation fixed

The Windows implementation of function OSAPI_Heap_realloc() had a precondition to check for a NULL pointer as input parameter. This is wrong as in this case the function shall allocate a new buffer (equivalent to malloc()).

This issue has been fixed.

[RTI Issue ID MICRO-1655]

Use API DDSDomainParticipant::delete_contained_entities() in C++ examples

Shipped C++ examples now use DDSDomainParticipant::delete_contained_entities() to delete all DSS entities in a DDS Participant. This is easier than using DDSDomainParticipant::delete_topic(), DDSDomainParticipant::unregister_type(), etc.

This issue has been fixed.

[RTI Issue ID MICRO-1656]

Memory leak in shipped examples fixed

Shipped examples were not releasing correctly some of the allocated structures when application finalized.

This issue has been fixed.

[RTI Issue ID MICRO-1676]

C++ shipped examples might release an object twice.

C++ shipped examples might release an object twice in case of error.

This issue has been fixed.

[RTI Issue ID MICRO-1677]

Backwards Compatability

Change in on_before_sample_deserialize callback.

In 2.4.10 and earlier the stream passed to on_before_sample_deserialize callback started at the encapsulation header followed by user data. However, with the added support for batched samples this is no longer possible. Instead the stream now starts at the user-data payload. Note that the only supported encapsulation format for user-data is CDR. This may change in future versions.

The examples have been updated to reflect the change. Please refer to the examples for details.

8.5.5 What's New in 2.4.10.1

UDP Transformations

This release includes UDP Transformations which enables regular UDP sockets to be used with custom payload transformations. Please refer to ref UserManuals_UDPTransform for details. The UDP Transformation feature is enabled by default in this release. However, future releases may disable the feature by default. Thus, it is advised to always compile with the UDP Transformation feature enabled (-DRTIME_UDP_ENABLE_TRANSFORMS=1 to cmake).

NOTE: In the the EAR for 2.4.10.1 the default behavior was to allow both plain UDP and transformed UDP traffic when transformations was compiled in. This has changed. The default is to disable regular UDP. In order to support it the transform_udp_mode must be set to UDP_TRANSFORM_UDP_MODE_ENABLED. Since this may change in future release it is advised to always set the correct mode of operation.

8.5.6 What's Fixed in 2.4.10.1

Race Condition when Log Buffer is Full and a Custom Log-handler is Installed

A race condition existed when a custom log handler was installed and the log buffer was full. A temporary message was created to hold the minimum log data and when the custom log handler was called it was possible that a new log entry was added while the custom log handler parsed the temporary message.

This has been fixed in this version.

[RTI Issue ID MICRO-1641]

8.5.7 What's New in 2.4.10

Generate Example Application with rtiddsgen

It is now possible to generate an example application for RTI Connext Micro using rtiddsgen. To generate an example:

:: rtiddsgen -language C | C++ -micro -example <IDL File>

A CMakeLists.txt file is generated that can be used with rtime-make:

:: rtime-make [options] –srcdir <path to CMakeLists.txt>

Please refer to the generated README.txt file for details.

BY_SOURCE_TIMESTAMP_DESTINATIONORDER Support on DataWriter

The DataReader and DataWriter Qos policy now includes the DDS_DestinationOrderQosPolicy:

- The DDS DataReader only supports BY_RECEPTION_DESTINATION_ORDER (the default value).
- The DDS DataWriter supports BY_RECEPTION_TIMESTAMP_DESTINATION_ORDER and BY_SOURCE_TIMESTAMP_DESTINATION_ORDER.

Please refer to the DDS reference manual for details.

[RTI Issue ID MICRO-1597]

8.5.8 What's Fixed in 2.4.10

Linker Warning for Missing PDB Files

The i86Win32VS2010 libraries shipped with *Connext DDS Micro* did not include PDB files. For this reason, when compiling an application a warning similar to the following may have been shown:

:: rti_mezd.lib(BuiltinTopicData.obj) : warning LNK4099: PDB 'dds_czd.pdb' was not found with 'rti_mezd.lib(BuiltinTopicData.obj)' or at '<path>\dds_czd.pdb'; linking object as if no debug info

The warning was harmless and only indicates that debug information was missing for the linked libraries.

[RTI Issue ID MICRO-1556]

Linking with Dynamic Windows C Run-Time (CRT)

All shipped *Connext DDS Micro* libraries for Windows platforms (static release/debug, dynamic release/debug) now link with the dynamic Windows C Run-Time (CRT). Previously, the static *Connext DDS Micro* libraries statically linked the CRT.

An existing Windows project that is linking with the *Connext DDS Micro* static libraries must update the RunTime Library settings.

In Visual Studio, select C/C++, Code Generation, Runtime Library, select:

- Multi-threaded DLL (/MD) instead of Multi-threaded (/MT) for static release libraries.
- Multi-threaded Debug DLL (/MDd) instead of Multi-threaded Debug (/MTd) for static debug libraries.

For command-line compilation, use:

- /MD instead of /MT for static release libraries.
- /MDd instead of /MTd for static debug libraries.

In addition, it may be necessary to ignore the static run-time libraries in their static configurations. In Visual Studio, select Linker, Input in the project properties and add libcmtd;libcmt to the 'Ignore Specific Default Libraries' entry.

For command-line linking, add /NODEFAULTLIB:"libcmtd" /NODEFAULTLIB:"libcmt" to the linker options.

[RTI Issue ID MICRO-1572]

DDS_Publisher_create_datawriter() May Fail to Create a New Datawriter

When an application reaches the local_writer_allocation resource limit, where subsequent calls to DDS_Publisher_create_datawriter() fail to create a new DataWriter, calling DDS_Publisher_delete_datawriter() should reclaim resources of the deleted DataWriter and allow the creation of a new DataWriter. However, in the previous release, in certain cases there was a problem with reclaiming DataWriter resources that prevented the creation of a new DataWriter.

Deleting a DataWriter or DataReader involves acknowledgements from matched applications. Thus, calling DDS_Publisher_delete_datawriter() is not an instantaneous operation so resources may not be available immediately. When this case occurs, calling DDS_Publisher_create_datawriter() after a short duration may be successful. The maximum time for a resource to be released is the maximum time a response is expected from a matched application based on the DPDE discovery plugin configuration for the built-in discovery endpoints.

[RTI Issue ID MICRO-1579]

DataReader May Not Reclaim NOT_ALIVE Instances when DataWriter Deleted or Liveliness Lost

Applications using read()/take() in on_data_available may not have received NOT_ALIVE_NO_WRITERS for instances that changed state to NOT_ALIVE_NO_WRITERS when a deleted data writer or data reader lost liveliness with a data writer caused the change. This has been fixed.

[RTI Issue ID MICRO-1580]

A Datawriter may fail to release instance resources if a peer is inactive while the Participant liveliness expires

A reliable DataWriter can mark a matched DataReader as inactive if the DataReader fails to respond to heartbeats, as configured by max_heartbeat_retries. However, if a DataReader is marked as inactive and the Participant liveliness for the DataReader's Participant expires, a DataWriter afterwards may have failed to reclaim instances resources if unregister_instance() was called. This has been fixed.

[RTI Issue ID MICRO-1581]

A Reliable DataWriter With max_samples_per_instance = 1 May Run Out of Resources After Multiple Unregistrations of Single Instance

A reliable DataWriter with max_samples_per_instance = 1 may have run out of instance resources if the same instance is unregistered multiple times before an acknowledgement is received from a matched DataReader. This has been fixed.

[RTI Issue ID MICRO-1583]

Connext Micro Fails to Discover Endpoints created by Connext Core if the Endpoints are Deleted or Modified

If an application developed with RTI Connext Core used set_qos() on an enabled endpoint or deleted and created new endpoints before *Connext DDS Micro* had discovered the deleted endpoints, *Connext DDS Micro* failed to discovery new endpoints. This has been fixed.

[RTI Issue ID MICRO-1588]

Incorrect Log Output in a Complete Log Message could not be Stored

If there was insufficient space to store a complete log-message, the default display function would incorrectly try to print log-data beyond the log-buffer. This has been fixed.

[RTI Issue ID MICRO-1589]

Possible Segmentation Fault when Unregistering TRANSIENT_LOCAL Instance

Calling unregister_instance() on the same TRANSIENT_LOCAL instance may have caused a segmentation fault. The segmentation fault occured when a call to unregister_instance() is acknowledged and a later call on unregister_instance() for the same instance had not been acknowledged yet. For the segmentation fault to occur there must be more than 1 call to unregister_instance() within the history depth. This has been fixed.

[RTI Issue ID MICRO-1590]

Support to map IDL modules to C++ namespaces in generated type-plugins

The rtiddsgen included by this release will correctly generate C++ code for data types defined within IDL modules, when passed the "-namespace" argument. Consider the following IDL:

```
module A {
  struct Foo {
    long bar;
  };
};

module B {
  struct Foo {
    long bar;
  };
};
```

C++ code generated by previous releases of rtiddsgen for this IDL input would fail to build if the "-namespace" argument was used to map each IDL module to a C++ namespace.

Some of the automatically generated data types were incorrectly being exported with C linkage, effectively disabling the C++ namespaces. This would cause duplicate symbols to be detected if two types with the same name were defined in two different modules.

[RTI Issue ID MICRO-1600]

Possible Memory Access Violation when Receiving Malformed RTPS Message

When a received RTPS message had its message and submessage headers processed, *Connext DDS Micro* incorrectly did not validate for all cases that there was sufficient space in the message's receive buffer before accessing a field of a header. Consequently, reception of certain malformed messages could have resulted in memory access violations. The problem has been fixed by always validating for sufficient buffer. This has been fixed.

[RTI Issue ID MICRO-1614]

In Some Cases an Incorrect Timeout Calculation Caused 100% CPU Load

Some combinations of timeouts, clock resolution and resource-limits may have caused an incorrect timeout to be scheduled causing an infinite loop in the timer thread.

If multiple timers expires at the same time and the timeout is exactly:

:: (dp_qos.resource_limits.remote_participant_allocation + (3*dp_qos.resource_limits.local_writer_allocation) + (3*dp_qos.resource_limits.local_reader_allocation) + 1) / 2 * timer_resolution

the next timeout may be scheduled for immediate timeout, causing the timer thread to consume excessive CPU.

[RTI Issue ID MICRO-1617]

8.5.9 What's New in 2.4.9

Improved Support for adding new Ports

Some changes were made to how *Connext DDS Micro* includes different ports. In versions before 2.4.9 new ports would typically update osapi_config.h and add a new directory with an implementation for the required OSAPI functions. As of 2.4.9 osapi_config.h was re-factored and OS and compiler specific functions were moved to two new files:

- osapi_os_<osname>.h This file contains OS specific information. RTI ships three files: osapi_os_posix.h, osapi_os_windows.h and osapi_os_vxworks.h. It is recommended to add a new osapi_os_<osname>.h file when a new OS is added.
- osapi_cc_<osname>.h This file contains compiler specific informations. RTI ship osapi_cc_stds.c which works with Microsoft Visual Studio, clang and GCC.

Please refer to ref OSAPIUserManuals_PortingModule for details.

Updated Build Environment to Build RTI Connext Micro

Connext DDS Micro now includes better support for adding CMake tool-chain files and also includes a better infrastructure to manage multiple builds of *Connext DDS Micro*. It is strongly encouraged to read ref OSAPIUserManuals_SourceModule for details to get familiar with the new build environment.

Example CMake Tool-chain Files for Cross-Compilation

Connext DDS Micro ships with a more cmake tool-chain files for Linux, Darwin, Windows and VxWorks. Please refer to ref OSAPIUserManuals_SourceModule for details.

[RTI Issue ID MICRO-706]

Host Bundle without the Java RunTime Available

A new smaller host bundle that does not include Java Runtime Environments (JRE) is now available for download. A host bundle with JREs included is still available.

With Java being necessary for the rtiddsgen utility, rtiddsgen now picks Java based on the following order:

- New rtiddsgen command line option -jre
- JREHOME environment variable
- JAVA_HOME environment variable
- JRE shipped with the host bundle
- PATH environment variable

[RTI Issue ID MICRO-1520]

Support for 64-bit Platforms

Connext DDS Micro was written for 32 bit architectures and is for all practical purposes a 32 bit application. There is no advantage to compiling *Connext DDS Micro* for a 64 bit architecture and the only reason to do so is if *Connext DDS Micro* must execute in a 64 bit environment for other reasons, such as other applications being 64 bit or 64 bit libraries not being available.

Connext DDS Micro is compiled and tested on various 64 bit architectures (iOS, MacOS, Windows, Linux, VxWorks). However, when doing so the following must be kept in mind:

- *Connext DDS Micro* does not work with any data-type larger than what the transport supports and up to a maximum of 2 GB.
- Timestamps in *Connext DDS Micro* are limited to seconds encoded as a signed 32 bit integer.

POSIX Compliance Improvements

Connext DDS Micro supports various POSIX like operating systems. Due to small differences in the implementations not all POSIX like are equal and OS specific adaptations are necessary.

As of 2.4.9 *Connext DDS Micro*'s POSIX OSAPI implementation conforms to:

- POSIX Std 1003.1, 2004 Edition (_POSIX_C_SOURCE 200112L)
- X/Open 6 (_XOPEN_SOURCE 600)

The *Connext DDS Micro* UDP transport uses ioctl calls to enable certain socket features. The required flags are in non-standard header-files on some operating system. In addition, not all POSIX-like operating systems support all the features. *Connext DDS Micro* checks which OS it is compiled for by testing the presence of preprocessor flags. As of 2.4.9 *Connext DDS Micro* has been built and tested on the following operating systems that supports a POSIX API (osapi_os.h):

- Linux (_linux_)
- Mac OS X (10.6 and later) ((_APPLE) && defined(MACH_))
- QNX 6.x (_QNXNTO_)
- VOS (_VOS_)
- iOS (((_APPLE) && defined(MACH_))
- Android (_linux_ && _ANDROID_)

NOTE: An additional compile option to enable certain non-POSIX features can be enabled to unchecking the RTIME_OSAPI_ENABLE_STRICT_POSIX option in the cmake-gui or by defining the C preprocessor flag -DOSAPI_ENABLE_STRICT_POSIX=1

C++ Support for find_topic()

The operation DDS_DomainParticipant_find_topic() is now natively supported by the C++ API as DDSDomainParticipant::find_topic().

Types Are Automatically Unregistered Upon Deleting Contained Entities

In previous releases, types must be unregistered manually from a DomainParticipant before the participant can be deleted. Now in this release, all registered types are automatically unregistered when calling DDS_DomainParticipant_delete_contained_entitites().

NOTE: It is legal to register the same type multiple times as long as it is registered with the same type-plugin. If manually unregistering a type, the type must be unregistered the same number of times as it was registered. DDS_DomainParticipant_delete_contained_entitites() ignores the number of times a type has been registered since all entities using a type are deleted first.

8.5.10 What's Fixed in 2.4.9

Improved Documentation

The *Connext DDS Micro* documentation has been improved for the following topics:

- Compiling the *Connext DDS Micro* source (ref OSAPIUserManuals_SourceModule)
- Filtering of samples by a DDS DataReader (ref UserManuals_MicroAndCore)
- How to use *Connext DDS Micro* with RTI Recorder (ref UserManuals_MicroAndCore)
- Compatibility between *Connext DDS Micro* and other RTI Products (ref UserManuals_MicroAndCore)

[RTI Issue ID MICRO-711, MICRO-1521, MICRO-1538, MICRO-1555]

Losing Participant Liveliness Stops Communication

Previously, given a DomainParticipant "P1" whose endpoints are communicating with other endpoints belonging to other DomainParticipants, when P1 detected liveliness lost with one other

DomainParticipant, communication incorrectly stopped with endpoints belonging to other DomainParticipants as well.

[RTI Issue ID MICRO-1543]

DDSTopic::narrow() Returned Incorrect Value in C++

The function lookup_topicdescription() returned a DDSTopicDescription that caused DDSTopic::narrow() to segmentation fault when this DDSTopicDescription was passed to other functions.

DDSTopic::narrow() now correctly returns a DDSTopic when passed a DDSTopicDescription found with lookup_topicdescription().

[RTI Issue ID MICRO-1544]

PRECONDITION_NOT_MET Returned by deleted_topic() When Topic Is Not Use

delete_topic() incorrectly returned PRECONDITION_NOT_MET if there where multiple references to it (for example via find_topic()). This has been corrected and delete_topic() now returns DDS_RETCODE_OK if there are multiple references, but the reference count can be decremented.

[RTI Issue ID MICRO-1545]

Instance Resources Not Reclaimed When Unregistered

When an instance is unregistered on the data writer that is best-effort with infinite deadline or using TRANSIENT_LOCAL durability, the data writer fails to free the resources being used. As a result, new instances cannot be written. This has been fixed and when an instance is unregistered all resources associated with the key is released.

[RTI Issue ID MICRO-1546]

Invalid Memory Read Reported in Log.c

Some memory profile tools reported an invalid read in Log.c. This was caused by an invalid pointer access when the log buffer was full and has been corrected.

[RTI Issue ID MICRO-1550]

Unsupported Functions When Compiling With RTI_CERT Has Been Removed From Generated Code

Code generated by rtiddsgen to support user data types has been updated to properly support compilation with the flag RTI_CERT. All unsupported operations (e.g. FooTypeSupport_delete_data) are now excluded when RTI_CERT is specified.

[RTI Issue ID MICRO-1558]

The HelloWorld_cert Example Now Compiles When Linked Against a Library Built With RTI_CERT

The HelloWorld_cert called functions that were not supported by libraries built with RTI_CERT. This has been corrected.

[RTI Issue ID MICRO-1561]

Hostnames Are No Longer Validated

Previously in *Connext DDS Micro* 2.4.6, a function to validate IP hostnames as defined by RFC-952 was added and called before passing them to the OS. However, this function was too restrictive and excluded valid service names. Hostname validation is now only done directly by the OS.

[RTI Issue ID MICRO-1563]

A Participant May Not Be Rediscovered In Case Of Asymmetric Liveliness Loss

This problem was only present when using dynamic discovery.

Consider two participants A and B. In the previous release, if A lost liveliness with B, but B did not lose liveliness with A, then A did not completely rediscover B when their connection was reestablished. The problem was that since B had not lost liveliness with A, when a connection was reestablished, B thought A was already up to date on endpoint discovery. Hence, A did not rediscover the endpoints in B. This release has fixed this issue.

[RTI Issue ID MICRO-1571]

A Non-keyed Endpoint Matches a Keyed Endpoint

When performing matching between A DataReader and DataWriter the entity kind was not checked. This means a keyed DataReader would match a non-keyed DataWriter and a non-keyed DataReader would match an keyed DataWriter.

This issue would can happen if two different IDLs files are used to create DataReaders and DataWriters of the same topic and type.

Note that *Connext DDS Micro* does not support type validation. If two (or more) IDLs are used to describe the same keyed type there is no check that the key-fields are the same. Thus, even with this issue resolved there are still potential pitfalls with multiple IDLs for the same type.

[RTI Issue ID MICRO-1574]

8.5.11 What's New in 2.4.8

2.4.8 is a maintenance release with no new features.

8.5.12 What's Fixed in 2.4.8

Consistent support for assignment operator in C++

The assignment operator for the DDS Qos, Qos policy and Status structures were not consistently supported. This has been fixed in this release as follows:

- All QoS structures support the default generated C++ assignment operator.
- All QoS policy structures support the default generated C++ assignment operator.
- All Status structures support the default generated C++ assignment operator.

In addition, all QoS structures support the == and != operators.

[RTI Issue ID MICRO-1541]

DPSE API renamed to avoid conflict with assert()

The DPSE C++ API had methods called assert. However, this conflicts with the C assert() macro. This has been fixed in this release by updating the DPSE C++ API to be inline with the C API. The new API is:

```
class DDSCPPDllExport DPSEDiscoveryPlugin
{
public:
  static DDS_ReturnCode_t
  RemoteParticipant_assert(DDSDomainParticipant *const participant,
                           const char *rem_participant_name);

  static DDS_ReturnCode_t
  RemotePublication_assert(DDSDomainParticipant * const participant,
                           const char *const rem_participant_name,
                           const struct DDS_PublicationBuiltinTopicData *const data,
                           NDDS_TypePluginKeyKind key_kind);

  static DDS_ReturnCode_t
  RemoteSubscription_assert(DDSDomainParticipant * const participant,
                            const char *const rem_participant_name,
                            const struct DDS_SubscriptionBuiltinTopicData *const data,
                            NDDS_TypePluginKeyKind key_kind);
};
```

[RTI Issue ID MICRO-1539]

8.5.13 What's New in 2.4.7

2.4.7 is a maintenance release with no new features.

8.5.14 What's Fixed in 2.4.7

Statuses are passed as pointers instead of references to DDSDomainParticipantListeners

The statuses in the DDSDomainParticipantListener methods are now passed by reference instead of by pointer.

[RTI Issue ID MICRO-1524]

Missing assignment operator = in RT_ComponentFactoryId

The C++ API did not include the assignment operator for the RT_ComponentFactoryId type. The following assignment operators have been added:

```
RT_ComponentFactoryId& operator=(const char *const name);
RT_ComponentFactoryId& operator=(const RT_ComponentFactoryId& from);
const RT_ComponentFactoryId& operator=(const RT_ComponentFactoryId& from) const;
```

[RTI Issue ID MICRO-1525]

CMAKE_C_FLAGS_ORIGINAL in CMakeLists.txt misspelled

The CMAKE_C_FLAGS_ORIGINAL variable in the CMakeLists.txt file was misspelled causing the original C_FLAGS to be ignored. This has been corrected in this release.

[RTI Issue ID MICRO-1526]

Missing const qualifier for the sequence [] operator

The C++ API was missing the const qualifier for the sequence [] operator. This has been corrected in this release with these operators:

```
T& operator[] (RTI_INT32 index);
const T& operator[] (RTI_INT32 index) const;
```

[RTI Issue ID MICRO-1527]

Missing primitive IDL sequences in C++

The C++ API did not include sequence of the primitive IDL types. This has been corrected in this release. Please refer to ref DDSUserManuals_SequenceModule for more information about the sequence API.

[RTI Issue ID MICRO-1529]

8.5.15 What's New in 2.4.6

Important API Changes

This version of *Connext DDS Micro* includes a number of API changes to improve compatibility with rticore and make the API more robust to input argument errors such as string length violations. Please note that some of the changes are incompatible with earlier version of *Connext DDS Micro*.

Changed and Incompatible APIs:

- DDS_SEQUENCE_INITIALIZER(t) has changed to DDS_SEQUENCE_INITIALIZER. That is, the sequence element type is no longer passed in.
- Foo_seq_get_contiguous_buffer replaces Foo_seq_get_buffer.
- DDSTopic now uses multiple inheritance. Thus, it is no longer necessary to explicitly convert a topic to a topic description with the as_topicdescription() method when creating calling create_datareader() in C++.
- The idref_DiscoveryComponent_name value has changed type from a char pointer to a RT_ComponentFactoryId_T type. Use ref RT_ComponentFactoryId_set_name to set the name of the discovery plugin name.
- All C++ statuses are passed as a const reference instead of a const pointer to the listeners.

New APIs:

- By default the full sequence API has been enabled. In previous versions only a limited subset was enabled. NOTE: For RTI_CERT the default sequence API is still the limited API.
- The following new sequence methods have been added to the full sequence API (excluding the DDSConditionSeq):
  - ensure_length
  - to_array
  - from_array
  - operator[] in C++ is equivalent to get_reference()
  - operator= is equivalent to _copy()
  - operator== is equivalent to _is_equal()
  - operator!= is equivalent to !_is_equal()
- The following new sequence methods have been added to the DDSConditionSeq:
  - ensure_length
  - operator[] in C++ is equivalent to get_reference()
  - operator= is equivalent to _copy()
  - operator== is equivalent to _is_equal()
  - operator!= is equivalent to !_is_equal()
- RTIBool has been added (it is used by rticore) and is equivalent to RTI_BOOL in *Connext DDS Micro*.
- A new method idref_EntityNameQosPolicy_set_name has been added to set the idref_EntityNameQosPolicy_name field.
- Please refer to ref rl_new_246_MICRO-1512 for new C++ APIs.

Run-time Memory Footprint Has Been Significantly Reduced

The internal representation of state information has been refactored, significantly reducing run-time memory usage.

Please refer to the ref DDSUserManuals_ResourceModule guide for details.

New FooTypeSupport operations

The FooTypeSupport code generated for a user-defined Foo data type now includes three additional operations:

- FooTypeSupport::get_type_name
- FooTypeSupport::create_data
- FooTypeSupport::delete_data

These operations are available to users of both the C and C++ APIs.

All public C API now natively available to C++ users

The missing parts of RTI Connext Micro's public C API have now been added to the public C++ API, so that C++ users don't have to rely on C operations to implement their applications.

C++ developers are also not required to include any C header file anymore, but they must instead rely on newly available C++ header files.

Please refer to ref CPPApiModule for a list of APIs.

Status data passed by reference to C++ listeners

All callbacks exposed by the DDS listeners of the C++ API (DDSDataReaderListener, DDSDataWriterListener, DDSTopicListener, and other derived classes) now accept the status data passed in by the middleware as a C++ reference, rather than a pointer.

TheParticipantFactory now available to C++ users

The variable TheParticipantFactory is now available to users of the C++ API to reference the singleton instance of DDSDomainParticipantFactory.

Status types now available in DDS:: C++ namespace

All the status types (e.g. DDS_SubscriptionMatchedStatus) have been exposed to C++ users as part of the DDS:: namespace (e.g. DDS::SubscriptionMatchedStatus).

Foo::copy_data() takes const argument

The pointer specifying the source sample passed to the generated operation Foo::copy_data() (C++ API) is now of "const" type.

ConditionSeq added to C++ DDS namespace

C++ developers can now refer to data type DDS_ConditionSeq as DDS::ConditionSeq.

First 2-Bytes Of GUID Assigned to Vendor ID

In order to be interoperable with the Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol (DDSI-RTPS), version 2.2, the first 2-bytes of every GUID are now automatically assigned to the OMG-specified Vendor ID.

8.5.16 What's Fixed in 2.4.6

POSIX Threads Were Created Without Names

Previous releases on POSIX platforms created threads with no names. In this release, if thread naming is supported, a POSIX thread created with the *Connext DDS Micro* OSAPI_Thread_new() function will have its thread name set.

[RTI Issue ID MICRO-638]

Prerequisite for HelloWorld_android updated in README.txt

The README.txt file for Android did not clarify that is it necessary to install the NDK tool-chain as a standalone toolchain. This has been fixed.

[RTI Issue ID MICRO-807]

CPP/HelloWorld_dpde example does not overwrite RTIMEHOME

In previous releases of *Connext DDS Micro*, the CPP/HelloWorld_dpde example overwrote the RTIMEHOME environment variable, making it impossible for developers to point it to any custom value.

This error was fixed and the example can now be compiled with any valid value of RTIMEHOME.

[RTI Issue ID MICRO-834]

Transport Not Supporting Multicast Did Not Ignore Multicast

Previously, if a multicast address was specified as a discovery or user_traffic address, it was not correctly ignored by transports that did not support multicast. Consequently, an application may have failed to create a DomainParticipant. This has been fixed in this release.

[RTI Issue ID MICRO-1153]

Discovery Messages Incorrectly Dropped When Containing Non-Standard Locators

When a discovery message was received with a non-standard locator, such as for an unsupported transport, rather than just ignore the locator, the entire discovery message was discarded. This incorrect behavior prevented discovery of the entity that sent the discovery message. This issue has been fixed in this release.

[RTI Issue ID MICRO-1270]

HEARTBEAT Not Sent in Response To Initial ACKNACK

In *Connext DDS Micro*, a newly matched reliable DataReader will send an initial ACKNACK submessage to the matching DataWriter in order to expedite reliable communication. The initial ACKNACK is zero-valued, and a DataWriter receiving it will not resend any samples but instead will send a HEARTBEAT that the DataReader will respond with a proper ACKNACK.

In the previous release, however, a DataWriter receiving this initial ACKNACK did not respond with a HEARTBEAT. Consequently, reliable resend of historical samples did not start as soon as it should have, and instead would start with the next HEARTBEAT sent by the DataWriter, either a periodic HEARTBEAT or a piggyback HEARTBEAT sent with newly written samples. This issue has been fixed in this release.

[RTI Issue ID MICRO-1443]

Incorrect Return Code From DataReader's Read or Take APIs When Max_Outstanding_Reads Exceeded

When a DataReader's read or take APIs are called, depending on the input parameters of the sample sequence and sample-info sequence, the DataReader may loan to the caller its memory containing sample and sample-info entries. A resource limit, DATA_READER_RESOURCE_LIMITS max_outstanding_reads, sets the maximum number of samples (and corresponding sample-info entries) that may be loaned.

In previous releases, when max_oustanding_reads was exceeded, the read/take APIs incorrectly returned DDS_RETCODE_NO_DATA instead of DDS_RETCODE_OUT_OF_RESOURCES. This bug has been fixed in this release.

[RTI Issue ID MICRO-1460]

DataReader Did Not Replace Historical Samples When max_samples_per_instance Equaled History Depth

Previously, given a DataReader with RESOURCE_LIMITS max_samples_per_instance equal to HISTORY depth, when the DataReader exceeded its depth (or max_samples_per_instance), it incorrectly did not replace the oldest historical sample with the newest sample. Instead, the oldest historical sample was kept in the queue, and subsequent calls to read() could return it. Note, calls to take() would remove all taken sample from the queue.

This issue has been fixed in this release.

[RTI Issue ID MICRO-1463]

A Disposed Instance Could Be Updated By A DataWriter That Is Not Its Exclusive Owner

When EXCLUSIVE_OWNERSHIP was used, a disposed instance could incorrectly be updated by a DataWriter with a lower strength than the DataWriter that disposed the instance, even if that DataWriter had not unregistered the instance. This has been corrected: when an instance is disposed, a lower strength DataWriter is not allowed to update the instance as long as the DataWriter that disposed the instance is still registered as an updater for the instance. Only when the DataWriter unregisters from the instance can a lower strength DataWriter update the instance again.

[RTI Issue ID MICRO-1464]

Fixed code generation for user-defined enum constants.

The previous version of rtiddsgen shipped with *Connext DDS Micro* contained a bug which prevented the numerical constants assigned to an enum's values to be correctly handled in the generated code.

This error has been fixed and IDL enum types are now correctly translated into C/C++ data types with the correct constants.

[RTI Issue ID MICRO-1483]

Hostname is verified as specified in RFC-952 and RFC-1123

Connext DDS Micro relied on gethostbyname() to resolve hostnames. However, if a name resolver was not available it was possible to specify illegal names.

This has been corrected and only legal names, as defined by RFC-952 and RFC-1123, are resolved.

[RTI Issue ID MICRO-1489]

DDS_<Foo>Seq APIs Were Missing

The DDS sequence APIs for the built-in DDS types, such as DDS_LongSeq etc, were missing. The workaround was to use CDR_<Foo>Seq instead.

This issue has been corrected in this release, with the missing sequence APIs now included.

[RTI Issue ID MICRO-1493]

DataReader Could Reject All Subsequent Samples From a DataWriter

In the previous release, given a DataReader receiving samples from a DataWriter, after the DataWriter had written approximately (2^32) - max_samples_per_remote_writer number of samples, no more samples from that DataWriter would be received by the DataReader. Instead, every subsequent sample from the DataWriter would be rejected. This was caused by an incorrect update of an internal counter of the DataReader.

[RTI Issue ID MICRO-1500]

POSIX Thread Priorities Not Changeable

It was not possible to change the priority of POSIX threads created in previous releases of *Connext DDS Micro*. Instead, a POSIX thread inherited the priority of its parent. This has been fixed in this release.

[RTI Issue ID MICRO-1502]

RTPS DATA Submessages with K-flag Set Were Dropped

Previously, RTPS DATA submessages with the K-flag set (indicating a serialized key payload) were not processed and instead dropped by a DataReader. This has been fixed and such DATA submessages are now processed and accepted.

[RTI Issue ID MICRO-1511]

8.6 Known Issues

8.6.1 Maximum Number of Components Limited to 58

The maximum number of components that can be registered is limited to 58.

8.6.2 CMake version 3.6 or Higher is Required to Build VxWorks with CMake

Limitations in CMake prior to 3.6 required forcing the compiler to a specific path. However, this resulted in warnings from CMake 3.6 and higher that this feature has been deprecated and instead the CMAKE_TRY_COMPILE_TARGET_TYPE should be used to prevent linking.

Unless there are specific needs,there are no plans to support CMake prior to 3.6 when building for VxWorks.

8.6.3 Endpoint Discovery Requires Unique Object IDs Across All Remote Endpoints

When using static endpoint discovery (DPSE), RTI Connext Micro requires that the object_id for statically asserted remote endpoints must be unique across all remote endpoints, as opposed to just between remote endpoints within the same participant. Note, this restriction was incorrectly documented as removed in version 2.4.1.

8.6.4 Compiler warnings on VxWorks

When compiling for VxWorks 6.9 with the -Wconversion flag there are compiler warnings of the type:

```
warning: conversion to 'DDS_Boolean' from 'int' may alter its value
```

These compiler warnings seem to be an issue with GCC for VxWorks and can be ignored. The problem is that returning a value from a expression seems to always be treated as an unbounded int as opposed to an int with a value of 0 or 1 as the C standard dictates.

8.6.5 OSAPI Does Not Always Detect Endianess

osapi_cc_stdc.h detects the CPU endianness by checking GCC predefined macros, such as ___BYTE_ORDER___. However, some versions of GCC does not set these macros, for example GCC for VxWorks. If osapi_cc_stdc.h does not find any of the flags, it incorrectly sets the CPU to little endian.

In this case it is _important_ that _one_ of the following preprocessor macros are defined:

- RTI_ENDIAN_BIG The CPU is big-endian
- RTI_ENDIAN_LITTLE The CPU is little-endian

NOTE: The VxWorks cmake toolchain file from RTI set these based on CPU type in the target name (–name option).

Chapter 9

Benchmarks

The benchmark section provides metrics for *Connext DDS Micro*. The information contained here is only meant as guidance and actual numbers will vary across different hardware and compilers. Please note that the numbers are generated before the final release and the source-code line count and library sizes may vary slightly. Performance numbers are always valid for the final release.

9.1 Latency Benchmarks

Latency measurements are provided for two different environments:

- *Xeon* – End-to-End latency measured on high-performance Xeon machines in a dedicated network using the RTI Connext DDS Performance Test tool.
- *Raspberry Pi* – Round-trip latencies measured on stock Raspberry Pi's in a large, non-dedicated network.

9.1.1 Xeon

The end-to-end latency is measured between two identical machines using the test configuration below and running the RTI Connext DDS Performance Test tool.

The test environment consists of:

- x86_64 CentOS Linux release 7.1.1503
- RTI Perftest 3.0
- Switch Configuration: D-Link DXS-3350 SR:
  - 176Gbps Switching Capacity
  - Dual 10-Gig stacking ports and optional 10-Gig uplinks
  - Stacks up to 8 units per stack
  - 4MB (Packet Buffer Size)
  - 48 x 10/100/1000BASE-T ports
- Machine:

  - Intel I350 Gigabit NIC
  - Intel Core i7 CPU:
    * 12MB cache
    * 6 Cores (12 threads)
    * 3.33 GHz CPU speed
  - 12GB memory

The latency is measured by sending one PING sample and wait for the Echoer to return the PONG sample. The sender records the time it took to receive the PONG sample and divides the result by 2. The test is repeated a number of times for each size. Note that the *end-to-end* latency is measured.

Interpretation of the measurements (all numbers are reported in micro-seconds):

- Bytes - The size of the DDS sample payload (UDP overhead is _not_ included) in bytes.
- Ave - Average latency
- Std - Standard deviation
- Min - The minimum latency
- Max - The maximum latency
- 50% - The 50th percentile latency
- 90% - The 90th percentile latency
- 99% - The 99th percentile latency
- 99.99% - The 99.99th percentile latency

C++ Best Effort keyed 1 Gbps

Bytes	Ave (us)	Std	Min (us)	Max (us)	50%	90%	99%	99.99%
32	29	0.6	27	91	29	29	30	33
64	28	0.7	27	328	28	29	30	34
128	30	0.6	28	52	30	30	31	35
256	33	0.6	31	285	33	33	35	38
1024	47	0.7	46	338	47	47	49	53
4096	80	0.6	79	272	80	81	82	86
8192	117	0.7	116	302	117	118	119	123
63000	609	1.0	606	630	608	610	611	624

C++ Best Effort Unkeyed 1 Gbps

Length	Ave (us)	Std	Min (us)	Max (us)	50%	90%	99%	99.99%
32	27	0.4	26	88	27	28	29	32
64	28	0.5	27	285	28	28	30	33
128	29	0.6	28	328	29	30	31	35
256	32	0.5	31	333	32	32	34	37
1024	46	0.8	45	345	46	47	48	52
4096	79	0.9	78	349	79	80	81	86
8192	116	0.9	115	335	116	117	119	123
63000	608	1.0	606	635	608	610	611	624

C++ Reliable Keyed 1 Gbps

Length	Ave (us)	Std	Min (us)	Max (us)	50%	90%	99%	99.99%
32	32	1.7	29	322	31	35	37	40
64	32	2.1	30	90	31	36	38	43
128	34	2.0	31	617	33	36	40	43
256	37	1.6	35	333	37	39	42	46
1024	52	1.6	50	414	52	54	57	61
4096	84	1.1	82	360	84	85	88	93
8192	122	1.9	120	604	121	123	126	131
63000	613	2.7	610	976	613	615	618	635

C++ Reliable Unkeyed 1 Gbps

Length	Ave (us)	Std	Min (us)	Max (us)	50%	90%	99%	99.99%
32	31	1.9	29	575	31	34	37	40
64	32	1.7	29	75	31	35	37	41
128	33	2.1	31	591	32	37	39	45
256	37	1.7	34	336	36	38	42	45
1024	51	1.5	48	328	51	53	56	60
4096	84	1.5	82	357	84	86	89	94
8192	121	1.4	119	412	121	123	126	130
63000	614	1.6	611	665	614	616	619	634

9.1.2 Raspberry Pi

The round-trip latencies are measured between two identical machines using the latency application available in the *Connext DDS Micro* example directory.

The test environment consists of:

- 2 x Raspberry Pi Model B+ with ARMv7 and 1 GB of memory
- Linux 4.14

- 1 Gbps network

Note that these tests are running on stock Raspberry Pis without any tuning for performance. In addition, these Raspberry Pis are part of a larger network used for scalability testing. Thus, the latency numbers provided here have a wider spread than the numbers in the dedicated *Xeon* test environment.

The latency is measured by sending one PING sample and wait for the Echoer to return the PONG sample. The sender then records the time it took to receive the PONG sample. The test is repeated a number of times for each size. Note that the *round-trip latency* is measured.

Interpretation of the measurements (all numbers are reported in micro-seconds):

- Bytes - The size of the DDS sample payload in bytes (UDP overhead is __not__ included)
- Mean - Average latency
- Min - The minimum latency
- 50% - The 50th percentile latency
- 90% - The 90th percentile latency
- 99% - The 99th percentile latency
- 99.99% - The 99.99th percentile latency

Round-trip Latency

Bytes	Mean (us)	Min (us)	50% (us)	90% (us)	99% (us)	99.99% (us)
16	1032.37	864.63	1010	1090	1370	6680
32	1045.11	910.63	1020	1090	1370	10500
64	1052.94	882.63	1030	1110	1380	9420
128	1096.95	915.63	1070	1150	1470	12000
256	1157.19	992.63	1130	1200	1470	9150
512	1294.60	1141.63	1260	1360	1660	7670
1024	1555.94	1401.63	1520	1640	1960	6440
2048	1964.19	1712.63	1930	2040	2380	13000
4096	2408.46	2109.63	2360	2500	2840	11200
8192	3181.26	2933.63	3120	3300	3660	12800
16384	4612.76	4337.63	4540	4700	5170	15000
32768	7762.30	7274.64	7740	7950	8420	20000

9.2 Throughput Benchmark

The throughput is measured between two identical machines using the throughput application available in the *Connext DDS Micro* example directory.

The test environment consists of:

- 2 Raspberry Pi Model B+ with ARMv7 and 1 GB of memory
- Linux 4.14

- 1 Gbps network

Interpretation of the measurements:

- Size - The size of the DDS sample payload (UDP overhead is not included)
- Demand - How many samples of the given size is written in a burst before a short delay (10ms). For example, a demand of 110 means that 110 samples are written in a burst followed by a 10ms delay before 110 samples is written again. This burst-delay-burst cycle continues for 1s before a new round is started.
- Samples - The number of samples written.
- Samples/sec - The number of samples written per second.
- Mbit/s - The bandwidth utilization for the payload based on Size and Samples/sec.
- Samples Lost - On the subscriber size the number of samples received is counted against what is expected.
- Samples Rejected - OLn the subscriber size the number of samples rejected by *Connext DDS Micro* is counted.
- CPU - The CPU load reached during a test. Note that how the CPU load is measured varies between platforms and is an approximation.
- Memory - The total amount of memory used during the test. This number should be constant as RTI Connext DDS Micro allocates all memory at creation time.

9.2.1 Publisher Throughput

Size	Demand	Samples	Samples/sec	Mbit/sec	Samples lost	Samples Rejected
16	10	33150	1657.33	0.212139	0	0
16	110	183370	9165.779775	1.17	0	0
16	210	257250	12859.12	1.65	0	0
16	310	210800	10528.63	1.35	0	0
32	10	33590	1679.19	0.43	0	0
32	110	204160	10203.62	2.61	0	0
32	210	241710	12079.59	3.10	0	0
32	310	269080	13450.27	3.44	0	0
64	10	34200	1709.68	0.88	0	0
64	110	203060	10149.83	5.20	0	0
64	210	254100	12699.31	6.5	0	0
64	310	208010	10399.80	5.32	0	0
128	10	33670	1683.31	1.72	0	0
128	110	184580	9228.79	9.4	0	0
128	210	232890	11641.43	11.92	0	0
128	310	214830	10728.35	10.99	0	0

9.2.2 Subscriber Throughput

Size	Demand	Samples	Samples/sec	Mbit/sec	Samples lost	Samples Rejected
16	10	33150	1657.23	0.21	0	0
16	110	183362	9167.67	1.17	8	0
16	210	33150	1657.23	0.21	0	0
16	310	33150	1657.23	0.21	0	0
32	10	33590	1679.00	0.21	0	0
32	110	204118	10202.81	1.17	8	0
32	210	240372	12010.05	0.21	0	0
32	310	266783	13338.10	0.21	0	0
64	10	34200	1710.48	0.88	0	0
64	110	202855	10139.21	5.19	205	0
64	210	251810	12585.23	6.44	2290	0
64	310	207959	10396.54	5.32	51	0
128	10	33670	1683.66	1.72	0	0
128	110	184410	9221.85	9.44	170	0
128	210	230254	11510.24	11.79	2636	0
128	310	213743	10675.63	10.93	108	0

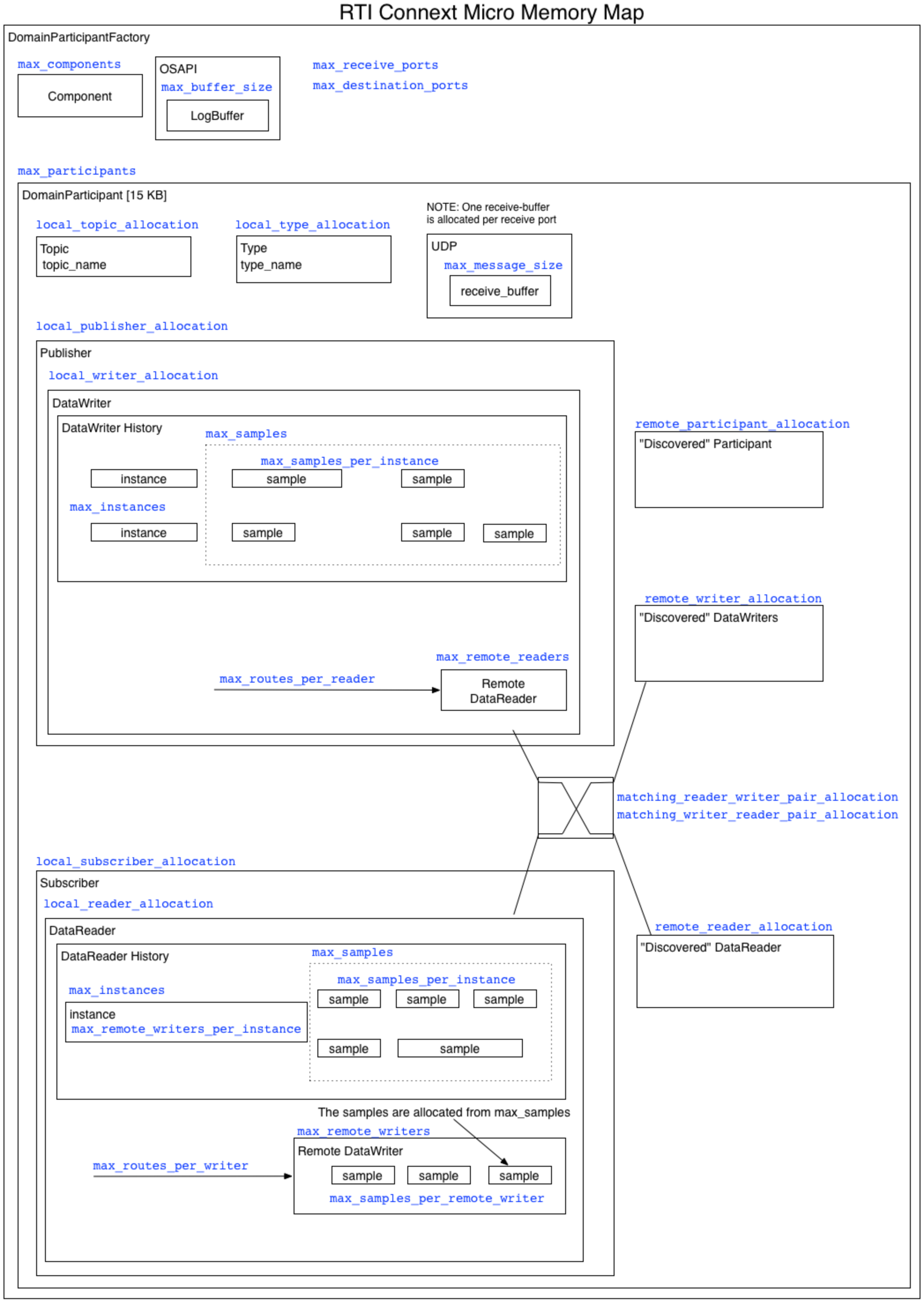
9.3 Heap Benchmarks

The “Heap” section provides information about how much dynamically allocated memory is used by *Connext DDS Micro*. It should be noted that exact numbers are very difficult to estimate and that the numbers are only for guidance. Please refer to ResourceModule for a more information on resource-limits and memory usage.

On Linux, for each heap allocation using malloc, malloc_usable_size() is called to determine the actual size of each allocation. The numbers include resources used by the RH_SM, WH_SM, and UDP components, but not the resources used by the dynamic discovery component (DPDE) or the static discovery component (DPSE). In addition, please note that the memory does not include memory for the actual user-data. This must be added according to the resource-limits. The numbers are for the release libraries.

The size for entities that are controlled by resource-limits are provided. In addition, a formula is provided to estimate the amount of memory used by a data reader and data writer as these are typically the ones that consume most of the memory.

9.3.1 Heap Usage



The following table shows how much memory each resource-limit in the memory model

Resource-limit	Size in Bytes	Notes
DomainParticipantFactory	2184	
max_participants	13712	This is the memory for an empty participar
max_components	N/A	
local_topic_allocation	116	Add strlen(topic_name) + 1
local_type_allocation	12	Add strlen(type_name) + 1
local_publisher_allocation	268	
local_subscriber_allocation	268	
local_reader_allocation	2184	The sample and instance resources must be
local_writer_allocation	2595	The sample and instance resources must be
matching_writer_reader_pair_allocation	28	
remote_participant_allocation	500	
remote_writer_allocation	about 600	This includes the topic_name
remote_reader_allocation	about 600	This includes the topic_name
max_destination_ports	77	
max_receive_ports	360	
(DataReader) max_instances	271	
(DataReader) max_samples	160	
(DataReader) max_remote_writers	391	
(DataReader) max_routes_per_writer	87	
(DataReader) max_samples_per_instance	0	
(DataReader) max_remote_writers_per_instance	0	
(DataReader) max_samples_per_remote_writer	0	
(DataWriter) max_instances	79	
(DataWriter) max_samples	116	
(DataWriter) max_remote_readers	391	
(DataWriter) max_routes_per_reader	87	
(DataWriter) max_samples_per_instance	0	
max_locators_per_discovered_participant	83	
max_buffer_size	0	
max_message_size	0	
matching_reader_writer_pair_allocation	0	

Calculating Memory Usage for DDS Entities

The following short-hands are used in these formulas:

- rl_ms - resource_limits.max_samples
- rl_mi - resource_limits.max_instances
- rl_mrw - datareader_resource_limits.max_remote_writers
- rl_mrpw - datareader_resource_limits.max_routes_per_writer
- wrl_mrw - datawriter_resource_limits.max_remote_readers
- wrl_mrpr - datawriter_resource_limits.max_routes_per_reader

Type

```
(13) + string.len(type_name) + 1
```

Topic

```
(117) + string.len(topic_name) + 1
```

DDS DataReader

```
(2184) + (rl_ms * 160 ) + (rl_mi * 271) + (rl_mrw * 391) + (rl_mrpw * 87)
```

DDS DataWriter

```
(2595) + (rl_ms * 116) + (rl_mi * 79) + (wrl_mrr * 391) + (wrl_mrpr * 87)
```

RemoteParticipant

```
(501) + (16 * 24)
```

RemotePublication

```
(149) + string.len(topic_name) + 1 + (16 * 24)
```

RemoteSubscription

```
(173) + strlen(topic_name) + 1 + (16 * 24)
```

9.3.2 Dynamic Discovery (DPDE) Heap Usage Information

The DPDE plugin is a DDS application that advertises locally created DDS entities and listens for DDS entities available in the DDS data-space. It is implemented using the DDS APIs supported by *Connext DDS Micro*.

The DPDE plugin creates the following DDS entities:

- One DDS Publisher
- One DDS Subscriber
- Three DDS Topics
- Three DDS DataReaders
- Three DDS DataWriters

The DPDE plugin also registers the following three types:

- DDS_ParticipantBuiltinTopicData
- DDS_PublicationBuiltinTopicData
- DDS_SubscriptionBuiltinTopicData

All heap memory allocated by the DPDE plugin is allocated after the DDS DomainParticipant is created (no additional memory is allocated after the DDS DomainParticipant is enabled).

DPDE Plugin	Release Size(B)
Plugin	65560

9.3.3 Static Discovery (DPSE) Heap Usage Information

The DPSE plugin is a DDS application that only advertises locally created DDS DomainParticipants and listens for other DDS DomainParticipants available in the DDS data-space. It is implemented using the DDS APIs supported by *Connext DDS Micro*.

The DPSE plugin creates the following DDS entities:

- One DDS Publisher
- One DDS Subscriber
- One DDS Topics
- One DDS DataReader
- One DDS DataWriter

The DPSE plugin also registers the following type:

- DDS_ParticipantBuiltinTopicData

All heap memory allocated by the DPSE plugin is allocated after the DDS DomainParticipant is created (no additional memory is allocated after the DDS DomainParticipant is enabled).

DSDE Plugin	Release Size(B)
Plugin	31644

9.4 Source Line Count

This section gives the size of each library in terms off effective lines of source-code (ELOC) and is gathered from the pre-processed files only for the release library. The ELOC number only include lines with source that directly contribute to the object-files. For example, the following are _not_ included:

- comments
- white-space
- lines with only braces
- type, structure, constant definitions

The ELOC number by itself is not very useful, but is provided since it is a frequently asked question.

Library	ELOC
rti_me	29102
discdpde	3334
discdpse	1648
rh_sm + wh_sm	1913

9.5 Library Sizes

The size of each shared library's _text_, and _data_ segment in bytes is determined using the *size* command on 64 bit Darwin. Please note that these numbers can vary significantly between different targets.

Library	Text (B)	Data (B)
rti_me	495616	495616
discdpde	53248	4096
discdpse	32768	4096
rh_sm	28672	4096
wh_sm	16384	4096
rti_me_cpp	90112	12288

9.6 Threads

RTI Connext DDS Micro uses multiple threads. The timer thread is managed by the domain participant and cannot easily be removed. All the UDP threads are managed by the UDP transport and a different UDP transport implementation can choose a different threading model.

Thread	Heap	Default Stack (1)
Timer	N/A	16384
UDP Receive	8192	16384

Notes:

1. The "Default Stack" is the stack size a thread is created with. It is _not_ the maximum stack size needed at run-time based on the deepest call-path.
2. This is the default maximum message size property. Each UDP thread allocates its own receive buffer.

Chapter 10

Copyrights

Printed in U.S.A. First printing.
May 2019.

Trademarks

Copy and Use Restrictions

Technical Support
Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://support.rti.com/

Chapter 11

Contact Support

We welcome your input on how to improve *RTI Connext DDS Micro* to suit your needs. If you have questions or comments about this release, please visit the RTI Customer Portal, https://support.rti.com. The RTI Customer Portal provides access to RTI software, documentation, and support. It also allows you to log support cases.

To access the software, documentation or log support cases, the RTI Customer Portal requires a username and password. You will receive this in the email confirming your purchase. If you do not have this email, please contact license@rti.com. Resetting your login password can be done directly at the RTI Customer Portal.

Chapter 12

Join the Community

RTI Community provides a free public knowledge base containing how-to guides, detailed solutions, and example source code for many use cases. Search it whenever you need help using and developing with RTI products.

RTI Community also provides forums for all RTI users to connect and interact.