RTI Connext DDS Micro

User's Manual

Version 3.0.0

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 Connext DDS and Connext Messaging.

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

RTI 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

RTI 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

RTI 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

RTI 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 RTI Connext DDS Documentation

The RTI Community site provides many resources for users of DDS and RTI. New users can start by going to RTI Documentation, downloading the RTI Core Libraries User's Manual for *RTI Connext DDS*, and reading Parts 1 (Introduction) and 2 (Core Concepts). These sections teach basic concepts of DDS applicable to all RTI middleware, including *RTI Connext DDS* and *RTI Connext DDS Micro*.

1.4 OMG DDS Specification

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

1.5 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 data bus 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 4 RTI target packages

- rti_connext_dds_micro-3.0.0-Unix.rtipkg
- rti_connext_dds_micro-3.0.0-Windows.rtipkg
- rti_connext_dds_micro_security_sdk-3.0.0-Unix.rtipkg
- rti_connext_dds_micro_security_sdk-3.0.0-Windows.rtipkg

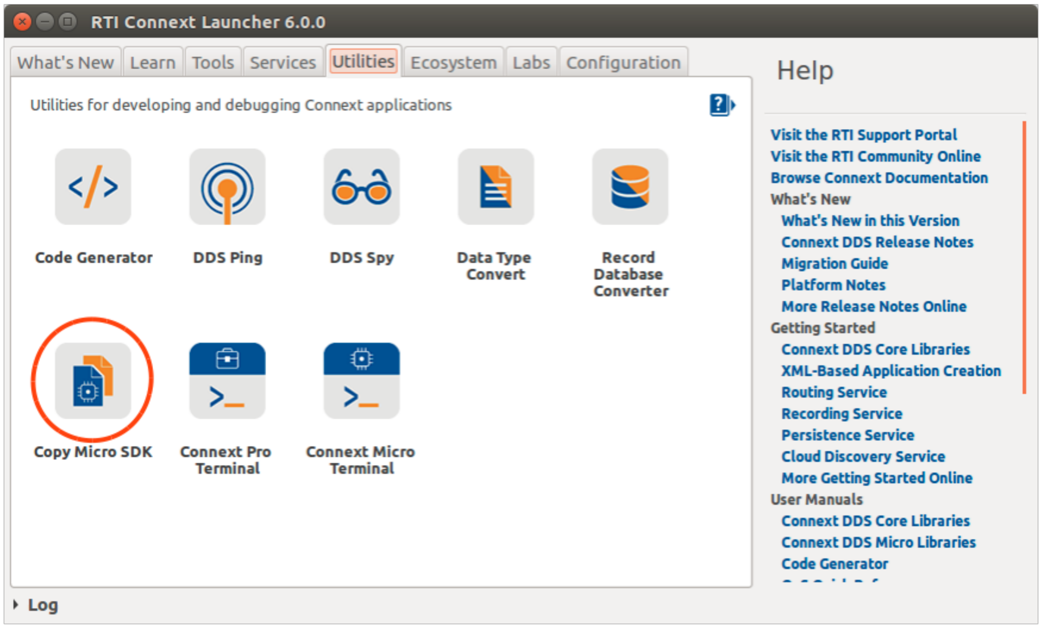
Note: you must first install *RTI Connext DDS* and install either `rti_connext_dds_micro-3.0.0-Unix.rtipkg` or `rti_connext_dds_micro-3.0.0-Windows.rtipkg` before the corresponding security SDK packages can be installed

Once installed, you will see a directory `rti_connext_dds_micro-3.0.0` in the *RTI Connext DDS* installation directory. This installation directory contains this documentation, the *rtiddsgen* code generation tool, and example source code. Note that a JRE is needed to execute *rtiddsgen*. A host bundle with JRE is shipped for build environments without JRE installed. A host bundle without JRE is also available for build environments that already have JRE installed.

It is strongly recommended that you copy the *RTI Connext DDS Micro* installation directory outside of the *RTI Connext DDS* installation. This is because it may not be desirable to build the *RTI Connext DDS Micro* libraries in *RTI Connext DDS* installation directory. To copy *RTI Connext DDS Micro* to another location, open up the *RTI Connext DDS* Launcher, navigate to the **Utilities** tab and click on the **Copy Micro SDK** and follow the instructions. See image below for a visual aid.

2.2 Setting Up Your Environment

The `RTIMEHOME` environment variable must be set to the installation directory path for *RTI Connext DDS Micro*. If you installed *RTI Connext DDS* with default settings, this will be in: `<path_to_connext_dds_installation>/rti_connext_dds-6.0.0/rti_connext_micro-3.0.0` If you have copied it to another place, set `RTIMEHOME` to point to that location.



2.3 Building RTI Connext DDS Micro

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

This section assumes 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. Unix:

```
cd <rti_me install directory>/src
# 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
```

   Windows:

```
cd <rti_me install directory>\src
# 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. The *RTI Connext DDS Micro* libraries are available in:

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

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

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

2.3.1 OpenSSL

The *Connext DDS Micro* builtin security plugin requires OpenSSL 1.0.1 or a later 1.0.x version, but is not compatible with 1.1 or later. The CMake build files will try to locate a suitable version and use a locally installed library if available. If a compatible library is not available, please check the RTI Download portal for a compatible version of OpenSSL. After installing OpenSSL, set OPENSSLHOME to its location when building.

```
rtime-make -DOPENSSLHOME=<path>/release
```

Excluding the Security Plugin from the Build

It is possible to exclude the builtin security plugin in *Connext DDS Micro* (the rti_me_seccore library). Set RTIME_TRUST_INCLUDE_BUILTIN to false to disable it.

```
rtime-make -DRTIME_TRUST_INCLUDE_BUILTIN=false
```

For help, use:

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

To list available targets use:

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

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

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

Chapter 3

Getting Started

3.1 Define a Data Type

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

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

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

As an example, the HelloWorld examples provided with *Connext DDS Micro* use this simple type defined in HelloWorld.idl. It contains a string "msg" with max length of 128 chars.

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*.

Calling the script to generate support code for 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 *RTI 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.

The generated code output consists of these files for 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 *DataWriter* and *DataReader*, and to register the type with a DDS *DomainParticipant*.

rtiddsgen does not generate publisher or subscriber code for *Connext DDS Micro*. This is different than for *Connext DDS*, where *rtiddsgen* will generate HelloWorld_publisher.c and HelloWorld_subscriber.c.

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 `-interpreted0` 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 Create an Application

The rest of this guide will walk you through the steps of creating an application and will provide example code snippets. It assumes that you have defined your types (see *Define a Data Type*) and have used *rtiddsgen* to generate their support code (see *Generate Type Support Code with rtiddsgen*).

3.3.1 Registry Configuration

The DomainParticipantFactory, in addition to its standard role of creating and deleting *DomainParticipants*, contains the RT Registry that a new application register with some necessary components.

The architecture of *RTI Connext DDS Micro* defines a run-time (RT) component interface that provides a generic framework for organizing and extending functionality of an application. An RT component is created and deleted with an RT component factory, and each RT component factory must be registered within an RT registry in order for its components to be usable by an application.

Connext DDS Micro automatically registers components that provide necessary functionality. These include components for DDS *Writers* and *Readers*, RTPS protocol, and the UDP transport.

In addition, every DDS application must register three components:

- **Writer History**. Queue of written samples of a *DataWriter*. Must be registered with name "wh".
- **Reader History**. Queue of received samples of a *DataReader*. Must be registered with name "rh".
- **Discovery (DPDE or DPSE)**. Discovery component. Choose either dynamic (DPDE) or static (DPSE) endpoint discovery.

Example source:

- Get the RT Registry from the DomainParticipantFactory singleton:

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

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

- Register the Writer History and Reader History components with the registry:

```
/* Register Writer History */
if (!RT_Registry_register(registry, "wh",
                          WHSM_HistoryFactory_get_interface(), NULL, NULL))
{
    /* failure */
}

/* Register Reader History */
if (!RT_Registry_register(registry, "rh",
```

(continues on next page)

(continued from previous page)

```
                          RHSM_HistoryFactory_get_interface(), NULL, NULL))
{
    /* failure */
}
```

Only one discovery component can be registered, either DPDE or DPSE. Each has its own properties that can be configured upon registration.

- Register DPDE for dynamic participant, dynamic endpoint discovery:

```
struct DPDE_DiscoveryPluginProperty discovery_plugin_properties =
    DPDE_DiscoveryPluginProperty_INITIALIZER;

/* Configure properties */
discovery_plugin_properties.participant_liveliness_assert_period.sec = 5;
discovery_plugin_properties.participant_liveliness_assert_period.nanosec = 0;
discovery_plugin_properties.participant_liveliness_lease_duration.sec = 30;
discovery_plugin_properties.participant_liveliness_lease_duration.nanosec = 0;

/* Register DPDE with updated properties  */
if (!RT_Registry_register(registry,
                          "dpde",
                          DPDE_DiscoveryFactory_get_interface(),
                          &discovery_plugin_properties._parent,
                          NULL))
{
    /* failure */
}
```

- Register DPSE for dynamic participant, static endpoint discovery:

```
struct DPSE_DiscoveryPluginProperty discovery_plugin_properties =
    DPSE_DiscoveryPluginProperty_INITIALIZER;

/*  Configure properties */
discovery_plugin_properties.participant_liveliness_assert_period.sec = 5;
discovery_plugin_properties.participant_liveliness_assert_period.nanosec = 0;
discovery_plugin_properties.participant_liveliness_lease_duration.sec = 30;
discovery_plugin_properties.participant_liveliness_lease_duration.nanosec = 0;

/* Register DPSE with updated properties  */
if (!RT_Registry_register(registry,
                          "dpse",
                          DPSE_DiscoveryFactory_get_interface(),
                          &discovery_plugin_properties._parent,
                          NULL))
{
    printf("failed to register dpse\n");
    goto done;
}
```

For more information, see the *Application Generation* section in the User's Manual.

3.4 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.

3.5 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 */
 }
```

3.5.1 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 */
}
```

3.5.2 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 */
}
```

3.5.3 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 discover. 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.

3.6 Create Publisher

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

In *RTI 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.

3.7 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");
    }
}
```

3.7.1 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 */
}
```

3.7.2 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.

3.8 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 *RTI 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.

3.9 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);
}
```

3.9.1 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.

3.9.2 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;
    }
}
```

3.9.3 Filtering Samples

In lieu of supporting Content-Filtered Topics, a DataReaderListener in *RTI 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.

3.10 Examples

RTI 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_dpde_secure** Same as the HelloWorld_dpde example, except that the RTI Security Plugins are installed and enabled on each DomainParticipant to perform mutual authentication, enforce access control rules, and encrypt data exchanged by applications.
- **HelloWorld_android**. Example application using Android NDK.
- **HelloWorld_static_udp**. Same as HelloWorld_dpde, except it uses static configuration of network interfaces.
- **HelloWorld_appgen**. Example application using Application Generation API.
- **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 *RTI Connext DDS Micro*.
- **Throughput**. Measures the end-to-end throughput of *RTI 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/Armv8). This process is commonly called *serialization/deserialization*, or *marshalling/demarshalling*.

RTI 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 3.9 Working with DDS Data Samples section of the RTI Connext DDS Core Libraries User's Manual.

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 the *RTI Code Generator's* **-sequenceSize** command-line argument (see the RTI Code Generator User's Manual).

Strings and Wide Strings

RTI 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 two-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 the *RTI Code Generator*'s **-stringSize** command-line argument (see the RTI Code Generator User's Manual).

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-16. This encoding shall be used as the wire format.

When the data representation is Extended CDR version 1, wide-string characters have a size of 4 bytes on the wire with UTF-16 encoding. When the data representation is Extended CDR version 2, wide-string characters have a size of 2 bytes on the wire with UTF-16 encoding.

Language bindings may use the representation that is most natural in that particular language. If this representation is different from UTF-16, the language binding shall manage the transformation to/from the UTF-16 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 2-byte integer. Users are responsible for using the right character encoding (UTF-16) when populating the wide-string values. This applies to all generated code, DynamicData, and Built-in data types. *RTI Connext DDS Micro* does not transform from the language binding encoding to the wire encoding.

Sending Type Information on the Network

RTI 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 User's Manual for more information.

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 *RTI 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 3.3 Creating User Data Types with IDL section in the RTI Connext DDS Core Libraries User's Manual for more information. Please note that not all features are supported by *Connext DDS Micro* as described in the following section.

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 *RTI 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 3.9 Working with DDS Data Samples section in the RTI Connext DDS Core Libraries User's Manual for more information.

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.

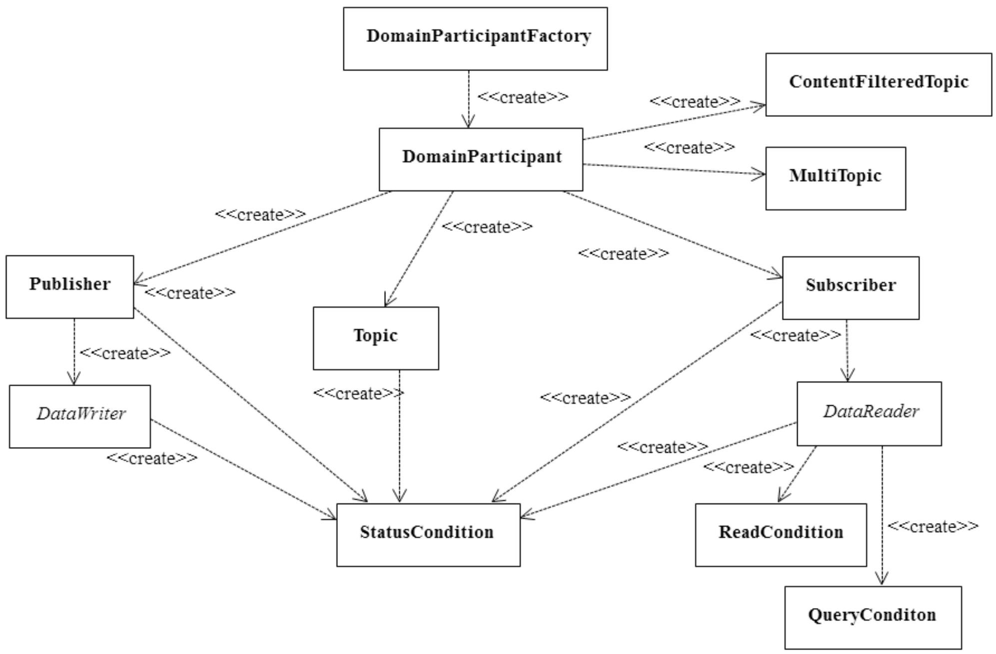


Figure 4.1: Overview of DDS Entities

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 Chapter 4 DDS Entities section of the RTI Connext DDS Core Libraries User's Manual. 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 HTML documentation.

For more details on creating, deleting, and setting up *DataWriters*, see the 6.3 DataWriters section of the RTI Connext DDS Core Libraries User's Manual.

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 HTML 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 7.2 Subscribers section of the RTI Connext DDS Core Libraries User's Manual.

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.

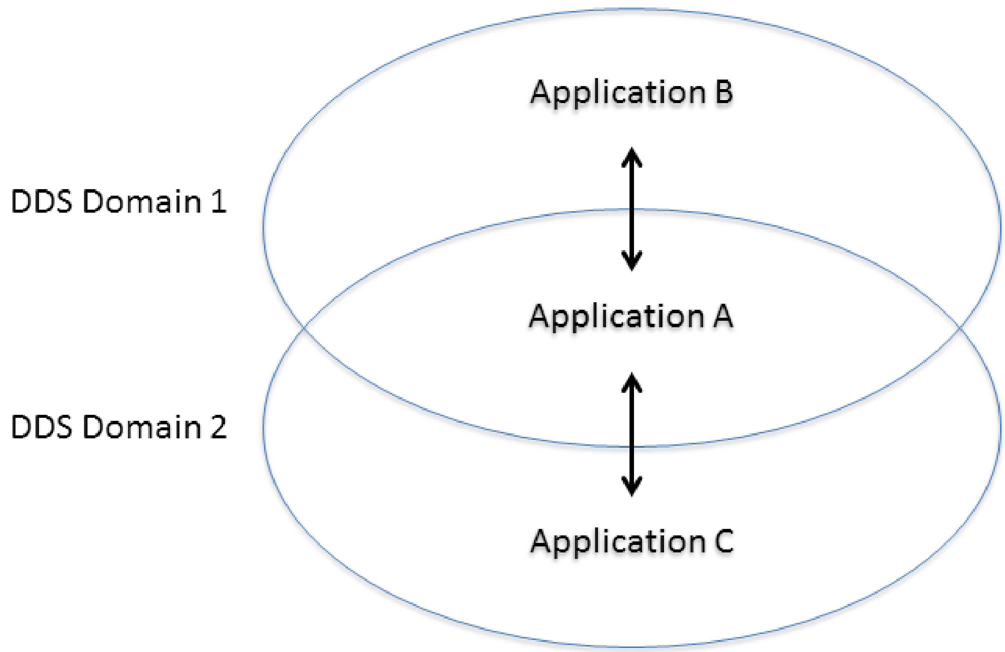


Figure 4.2: Relationship between Applications and DDS Domains
Applications can belong to multiple DDS domains—*A* belongs to DDS domains 1 and 2. Applications in the same DDS domain can communicate with each other, such as *A* and *B*, or *A* and *C*. Applications in different DDS domains, such as *B* 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*).

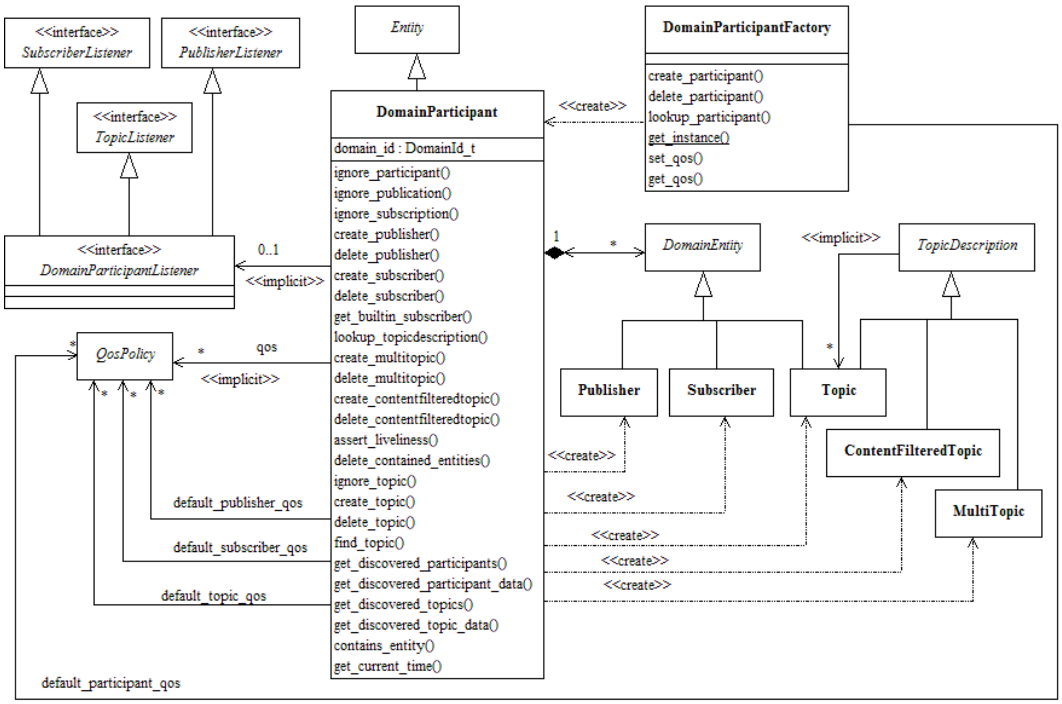


Figure 4.3: DDS Domain Module
Note: MultiTopics are not supported.

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 Application Generation

4.6.1 Introduction

RTI Connext DDS Micro's Application Generation feature simplifies and accelerates application development by enabling the creation of DDS *Entities* by compiling an XML configuration file, linking the result to an application, and calling a single API. Once created, all *Entities* can be retrieved from the application code using standard "lookup_by_name" operations so that they can be used to read and write data. C or C++ source code is generated from the XML configuration and compiled with the application.

This user-guide explains how to use this feature in an application and is organized as follows:

- *Overview*
- *Names Assigned to Entities*
- *Create a Domain Participant*
- *Retrieving Entities*
- *Interoperability*
- *Example Code*
- *Example Configuration*

4.6.2 Overview

The *Connext DDS Micro* Application Generation feature enables the creation of all DDS *Entities* needed in an application and the registration of the factories used in the application. Once created, all *Entities* can be retrieved from application code using standard "lookup_by_name" operations so that they can be used to read and write data. UDP transport, DPDE (Dynamic Participant Dynamic Endpoint) and DPSE (Dynamic Participant Static Endpoint) discovery configuration can also be configured as needed.

C source code is generated from the XML configuration and has to be compiled with the application. This is needed because *Connext DDS Micro* does not include an XML parser (this would significantly increase code size and amount of memory needed). The generated C source code contains the same information as the XML configuration file. The generated C source code can be used from both the C API Reference and C++ API Reference.

The *Connext DDS Micro* Application Generation is enabled by default in this release when compiling with rtime-make. However, future releases may disable the feature by default. Thus, it is advised to always compile with the *Connext DDS Micro* Application Generation feature enabled (-DRTIME_DDS_ENABLE_APPGEN=1 to CMake).

Important Points

Applications can create a *RTI Connext DDS Micro Entity* from a *DomainParticipant* configuration described in the XML Configuration file. All the *Entities* defined by such *DomainParticipant* configuration are created automatically as part of the creation. In addition, multiple *DomainParticipant* configurations may be defined within a single XML configuration file.

All the *Entities* created from a *DomainParticipant* configuration are automatically assigned an entity name. *Entities* can be retrieved via "lookup_by_name" operations specifying their name. Each *Entity* stores its own name in the QoS policies of the *Entity* so that they can be retrieved locally (via a lookup) and communicated via discovery.

A configuration file is not tied to the application that uses it. Different applications may run using the same configuration file. A single file may define multiple *DomainParticipant* configurations. Normally, a single application will instantiate only one *Connext DDS Micro*, but a *Connext DDS Micro* application can instantiate as many as needed.

Changes in the XML configuration file require to generate C/C++ source code again and recompile the application.

4.6.3 Names Assigned to Entities

Each *Entity* configured in the configuration is given a name. This name is used to retrieve them at run-time using the *RTI Connext DDS Micro* API.

In the context of the configuration we should distinguish between two names:

- Configuration name: The name of a specific *Entity*'s configuration. It is given by the name attribute of the corresponding element.
- Entity name in the *Entity*'s QoS: The Entity name in the *Entity*'s QoS.

At runtime, the *Entity* will be created using the Entity name in the *Entity*'s QoS; the configuration name will be used if this is an empty string.

The attribute multiplicity indicates that a set of *Entities* should be created from the same configuration. As each *Entity* must have a unique name, the system will automatically append a number to the Entity name in the *Entity*'s QoS (or, if it is an empty string, the configuration name) to obtain the Entity name. For example, if we specified a multiplicity of "N", then for each index "i" between 0 and N-1, the system will assign Entity names according to the table below:

Entity Name	Index: i
"configuration_name"	0
"configuration_name#i"	[1,N-1]

That is, the *Entity* name followed by the token "#" and an index.

4.6.4 Create a Domain Participant

To create a *DomainParticipant* from a configuration profile, use API create_participant_from_config(), which receives the configuration name and creates all the *Entities* defined by that configuration. This API is available in *RTI Connext DDS Micro* for compatibility with *RTI Connext DDS Professional*.

4.6.5 Retrieving Entities

After creation, you can retrieve the defined *Entities* by using the lookup_by_name() operations available in the C API Reference and C++ API Reference.

4.6.6 Interoperability

Applications created using this feature can inter-operate with other *RTI Connext DDS Micro* applications which are not created using this feature and with *RTI Connext DDS Professional* applications.

4.6.7 Example Code

This section contains an example to instantiate an application using *Connext DDS Micro* Application Generation.

Create the application

Create an application using *Connext DDS Micro* Application Generation. Note that only the *Connext DDS Micro* Application Generation factory needs to be registered; all other factories, such as UDP transport, DPDE, and DPSE discovery can be defined in the *Connext DDS Micro* Application Generation configuration, and are automatically registered by *Connext DDS Micro* Application Generation.

```
DDS_ReturnCode_t retcode;
DDS_DomainParticipantFactory *factory = NULL;
RT_Registry_T *registry = NULL;
struct APPGEN_FactoryProperty model_xml = APPGEN_FactoryProperty_INITIALIZER;
DDS_DomainParticipant *participant;
DDS_DataWriter *datawriter;
struct DDS_DataWriterListener dw_listener =
    DDS_DataWriterListener_INITIALIZER;
factory = DDS_DomainParticipantFactory_get_instance();
registry = DDS_DomainParticipantFactory_get_registry(factory);
/* This pointer must be valid as long as the Micro
   Application Generation plugin is registered
```

(continues on next page)

(continued from previous page)

```
 */
model_xml._model = APPGEN_get_library_seq();
/* Register factory used to create participants from config */
if (!APPGEN_Factory_register(registry, &model_xml))
{
    printf("failed to register Application Generation\n");
    goto error;
}
/* create participant from config */
participant = DDS_DomainParticipantFactory_create_participant_from_config(
                  factory, "UnitTestAppLibrary::UnitTestPublisherApp");
if (participant == NULL)
{
    printf("failed to create participant\n");
    goto error;
}
datawriter = DDS_DomainParticipant_lookup_datawriter_by_name(
                 participant, "TestPublisher1::Test1DW");
if (datawriter == NULL)
{
    printf("failed to lookup datawriter\n");
    goto error;
}
dw_listener.on_publication_matched = HelloWorldPublisher_on_publication_matched;
retcode = DDS_DataWriter_set_listener(datawriter, &dw_listener,
                                      DDS_PUBLICATION_MATCHED_STATUS);
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to set writer listener\n");
    goto done;
}
retcode = DDS_Entity_enable(DDS_DomainParticipant_as_entity(participant));
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to enable entity\n");
}
hw_datawriter = HelloWorldDataWriter_narrow(datawriter);
/* Using variable hw_datawriter call HelloWorldDataWriter_write()
   to write samples.
 */
```

Delete the application

Connext DDS Micro Application Generation does not include any new API that can be used to delete an application. Instead, the already existing APIs can be used.

```
DDS_ReturnCode_t retcode;
RT_Registry_T *registry = NULL;
DDS_DomainParticipantFactory *factory = NULL;
factory = DDS_DomainParticipantFactory_get_instance();
registry = DDS_DomainParticipantFactory_get_registry(factory);
```

(continues on next page)

(continued from previous page)

```
retcode = DDS_DomainParticipant_delete_contained_entities(participant);
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to delete contained entities: %d\n", retcode);
    return;
}
retcode = DDS_DomainParticipantFactory_delete_participant(participant);
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to delete participant: %d\n", retcode);
    return;
}
if (!APPGEN_Factory_unregister(registry, NULL))
{
    printf("failed to unregister Application Generation\n");
}
retcode = DDS_DomainParticipantFactory_finalize_instance();
if (retcode != DDS_RETCODE_OK)
{
    printf("failed to finalize domain participant factory: %d\n", retcode);
    return;
}
```

4.6.8 Example Configuration

This section contains an example configuration. The example code configuration has been generated from the example XML configuration files.

The example configuration defines one library named “HelloWorldAppLibrary”. This library defines four *RTI Connext DDS Micro* applications: one with a publisher and one with a subscriber, both using DPDE discovery, and one with a publisher and one with a subscriber, both using DPSE discovery. Applications using DPDE discovery are compatible and are able to communicate. Applications using DPSE discovery are compatible and are able to communicate.

Domain Participant “HelloWorldDPDEPubDP”

This application defines a publisher which uses DPDE discovery.

The application has one named “HelloWorldDPDEPubDP”, one named “HelloWorldDPDEPub”, and one named “HelloWorldDPDEDW” which uses topic name “Example HelloWorld”. The application registers one type with name “HelloWorld” and defines one with name “Example HelloWorld” which uses the type “HelloWorld”.

Domain Participant “HelloWorldDPDESubDP”

This application defines a subscriber which uses DPDE discovery.

The application has one named “HelloWorldDPDESubDP”, one named “HelloWorldDPDESub”, and one named “HelloWorldDPDEDR” which uses topic name “Example HelloWorld”. The application registers one type with name “HelloWorld” and defines one with name “Example HelloWorld” which uses the type “HelloWorld”.

Domain Participant "HelloWorldDPSEPubDP"

This application defines a publisher which uses DPSE discovery.

The application has one named "HelloWorldDPSEPubDP", one named "HelloWorldDPSEPub", and one named "HelloWorldDPSEDW" which uses topic name "Example HelloWorld" and has RTPS id 100. The application registers one type with name "HelloWorld" and defines one with name "Example HelloWorld" which uses type "HelloWorld".

The application asserts one remote participant named "HelloWorldDPSESubDP" and one remote subscription with ID 200, type name "HelloWorld", and topic name "Example HelloWorld".

Domain Participant "HelloWorldDPSESubDP"

This application defines a subscriber which uses DPSE discovery.

The application has one named "HelloWorldDPSESubDP", one named "HelloWorldDPSESub", and one named "HelloWorldDPSEDR" which uses topic name "Example HelloWorld" and has RTPS id 200. The application registers one type with name "HelloWorld" and defines one with name "Example HelloWorld" which uses the type "HelloWorld".

The application asserts one remote participant named "HelloWorldDPSEPubDP" and one remote subscription with ID 100, type name "HelloWorld", and topic name "Example HelloWorld".

Configuration Files

Example *Connext DDS Micro* Application Generation configuration file HelloWorld.xml:

```
<?xml version="1.0"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://community.rti.com/schema/6.0.0/rti_dds_
↪profiles.xsd">
    <!-- Type Definition -->
    <types>
        <const name="MAX_NAME_LEN" type="long" value="64"/>
        <const name="MAX_MSG_LEN"  type="long" value="128"/>
        <struct name="HelloWorld">
            <member name="sender"  type="string"  stringMaxLength="MAX_NAME_LEN"  key=
↪"true"/>
            <member name="message" type="string"  stringMaxLength="MAX_MSG_LEN"/>
            <member name="count"   type="long"/>
        </struct>
    </types>
    <!-- Domain Library -->
    <domain_library name="HelloWorldLibrary">
        <domain name="HelloWorldDomain" domain_id="0">
            <register_type name="HelloWorldType" type_ref="HelloWorld">
            </register_type>
            <topic name="HelloWorldTopic" register_type_ref="HelloWorldType">
                <registered_name>HelloWorldTopic</registered_name>
            </topic>
        </domain>
    </domain_library>
```

(continues on next page)

(continued from previous page)

```
    <!-- Participant Library -->
    <domain_participant_library name="HelloWorldAppLibrary">
        <domain_participant name="HelloWorldDPDEPubDP"
            domain_ref="HelloWorldLibrary::HelloWorldDomain">
            <publisher name="HelloWorldDPDEPub">
                <data_writer topic_ref="HelloWorldTopic" name="HelloWorldDPDEDW">
                    <datawriter_qos base_name="QosLibrary::DPDEProfile"/>
                </data_writer>
            </publisher>
            <participant_qos base_name="QosLibrary::DPDEProfile"/>
        </domain_participant>
        <domain_participant name="HelloWorldDPDESubDP"
            domain_ref="HelloWorldLibrary::HelloWorldDomain">
            <subscriber name="HelloWorldDPDESub">
                <data_reader topic_ref="HelloWorldTopic" name="HelloWorldDPDEDR">
                    <datareader_qos base_name="QosLibrary::DPDEProfile"/>
                </data_reader>
            </subscriber>
            <participant_qos base_name="QosLibrary::DPDEProfile"/>
        </domain_participant>
        <domain_participant name="HelloWorldDPSEPubDP"
            domain_ref="HelloWorldLibrary::HelloWorldDomain">
            <publisher name="HelloWorldDPSEPub">
                <data_writer topic_ref="HelloWorldTopic" name="HelloWorldDPSEDW">
                    <datawriter_qos base_name="QosLibrary::DPSEProfile"/>
                </data_writer>
            </publisher>
            <participant_qos base_name="QosLibrary::DPSEProfile"/>
        </domain_participant>
        <domain_participant name="HelloWorldDPSESubDP"
            domain_ref="HelloWorldLibrary::HelloWorldDomain">
            <subscriber name="HelloWorldDPSESub">
                <data_reader topic_ref="HelloWorldTopic" name="HelloWorldDPSEDR">
                    <datareader_qos base_name="QosLibrary::DPSEProfile"/>
                </data_reader>
            </subscriber>
            <participant_qos base_name="QosLibrary::DPSEProfile"/>
        </domain_participant>
    </domain_participant_library>
</dds>
```

Example QoS configuration file HelloWorldQos.xml:

```
<?xml version="1.0"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://community.rti.com/schema/6.0.0/rti_dds_
↪profiles.xsd">
    <qos_library name="QosLibrary">
        <qos_profile name="DefaultProfile" is_default_participant_factory_profile="true">
            <!-- Participant Factory Qos -->
            <participant_factory_qos>
```

(continues on next page)

(continued from previous page)

```
                <entity_factory>
                    <autoenable_created_entities>false</autoenable_created_entities>
                </entity_factory>
            </participant_factory_qos>
            <!-- Participant Qos -->
            <participant_qos>
                <discovery>
                    <accept_unknown_peers>false</accept_unknown_peers>
                    <initial_peers>
                        <element>127.0.0.1</element>
                        <element>239.255.0.1</element>
                    </initial_peers>
                    <enabled_transports>
                        <element>udpv4</element>
                    </enabled_transports>
                    <multicast_receive_addresses>
                        <element>udpv4://127.0.0.1</element>
                        <element>udpv4://239.255.0.1</element>
                    </multicast_receive_addresses>
                </discovery>
                <default_unicast>
                    <value>
                        <element>
                            <transports>
                                <element>udpv4</element>
                            </transports>
                        </element>
                    </value>
                </default_unicast>
                <transport_builtin>
                    <mask>UDPv4</mask>
                </transport_builtin>
                <resource_limits>
                    <local_writer_allocation>
                        <max_count>1</max_count>
                    </local_writer_allocation>
                    <local_reader_allocation>
                        <max_count>1</max_count>
                        </local_reader_allocation>
                    <local_publisher_allocation>
                        <max_count>1</max_count>
                    </local_publisher_allocation>
                    <local_subscriber_allocation>
                        <max_count>1</max_count>
                        </local_subscriber_allocation>
                    <local_topic_allocation>
                        <max_count>1</max_count>
                        </local_topic_allocation>
                    <local_type_allocation>
                        <max_count>1</max_count>
                        </local_type_allocation>
```

(continues on next page)

(continued from previous page)

```
                <remote_participant_allocation>
                    <max_count>8</max_count>
                    </remote_participant_allocation>
                <remote_writer_allocation>
                    <max_count>8</max_count>
                </remote_writer_allocation>
                <remote_reader_allocation>
                    <max_count>8</max_count>
                </remote_reader_allocation>
                <max_receive_ports>32</max_receive_ports>
                <max_destination_ports>32</max_destination_ports>
            </resource_limits>
        </participant_qos>
        <!-- DataWriter Qos -->
        <datawriter_qos>
            <history>
                <depth>32</depth>
            </history>
            <resource_limits>
                <max_instances>2</max_instances>
                <max_samples>64</max_samples>
                <max_samples_per_instance>32</max_samples_per_instance>
            </resource_limits>
            <reliability>
                <kind>RELIABLE_RELIABILITY_QOS</kind>
            </reliability>
            <protocol>
                <rtps_reliable_writer>
                    <heartbeat_period>
                        <nanosec>250000000</nanosec>
                        <sec>0</sec>
                    </heartbeat_period>
                </rtps_reliable_writer>
            </protocol>
            <!-- transports -->
            <unicast>
                <value>
                    <element>
                        <transports>
                            <element>udpv4</element>
                        </transports>
                    </element>
                </value>
            </unicast>
        </datawriter_qos>
        <!-- DataReader Qos -->
        <datareader_qos>
            <history>
                <depth>32</depth>
            </history>
            <resource_limits>
```

(continues on next page)

(continued from previous page)

```
                    <max_instances>2</max_instances>
                    <max_samples>64</max_samples>
                    <max_samples_per_instance>32</max_samples_per_instance>
                </resource_limits>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <reader_resource_limits>
                    <max_remote_writers>10</max_remote_writers>
                    <max_remote_writers_per_instance>10</max_remote_writers_per_instance>
                </reader_resource_limits>
                <!-- transports -->
                <unicast>
                    <value>
                        <element>
                            <transports>
                                <element>udpv4</element>
                            </transports>
                        </element>
                    </value>
                </unicast>
                <multicast>
                    <value>
                        <element>
                            <receive_address>127.0.0.1</receive_address>
                            <transports>
                                <element>udpv4</element>
                            </transports>
                        </element>
                    </value>
                </multicast>
            </datareader_qos>
        </qos_profile>
        <qos_profile name="DPDEProfile" base_name="DefaultProfile">
            <participant_qos>
                <discovery_config>
                    <builtin_discovery_plugins>SDP</builtin_discovery_plugins>
                </discovery_config>
            </participant_qos>
        </qos_profile>
        <qos_profile name="DPSEProfile" base_name="DefaultProfile">
            <participant_qos>
                <discovery_config>
                    <builtin_discovery_plugins>DPSE</builtin_discovery_plugins>
                </discovery_config>
            </participant_qos>
        </qos_profile>
    </qos_library>
</dds>
```

Generated source files

Example generated header configuration HelloWorldAppgen.h:

```
/*
WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.
This file was generated from HelloWorld.xml using "rtiddsmag."
The rtiddsmag tool is part of the RTI Connext distribution.
For more information, type 'rtiddsmag -help' at a command shell
or consult the RTI Connext manual.
*/
#include "HelloWorldPlugin.h"
#include "app_gen/app_gen.h"
#include "netio/netio_udp.h"
#include "disc_dpde/disc_dpde_discovery_plugin.h"
#include "disc_dpse/disc_dpse_dpsediscovery.h"
#define RTI_APP_GEN___udpv4__HelloWorldAppLibrary_HelloWorldDPDEPubDP_udp1 \
{ \
    NETIO_InterfaceFactoryProperty_INITIALIZER, \
    REDA_StringSeq_INITIALIZER, /* allow_interface */ \
    REDA_StringSeq_INITIALIZER, /* deny_interface */ \
    262144, /* max_send_buffer_size */ \
    262144, /* max_receive_buffer_size */ \
    8192, /* max_message_size */ \
    -1, /* max_send_message_size */ \
    1, /* multicast_ttl */ \
    UDP_NAT_INITIALIZER \
    UDP_InterfaceTableEntrySeq_INITIALIZER, /* if_table */ \
    NULL, /* multicast_interface */ \
    DDS_BOOLEAN_TRUE, /* is_default_interface */ \
    DDS_BOOLEAN_FALSE, /* disable_auto_interface_config */ \
    {   /* recv_thread */ \
        OSAPI_THREAD_USE_OSDEFAULT_STACKSIZE, /* stack_size */ \
        OSAPI_THREAD_PRIORITY_NORMAL, /* priority */ \
        OSAPI_THREAD_DEFAULT_OPTIONS /* options */ \
    },  \
    RTI_FALSE /* transform_locator_kind */ \
    UDP_TRANSFORMS_INITIALIZER \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=145, columnNumber=35 */
↪
#define RTI_APP_GEN___dpde__HelloWorldAppLibrary_HelloWorldDPDEPubDP_dpde1 \
{ \
    RT_ComponentFactoryProperty_INITIALIZER, /* _parent */ \
    {   /*participant_liveliness_assert_period */ \
        30L, /* sec */ \
        0L /* nanosec */ \
    }, \
    {   /*participant_liveliness_lease_duration */ \
        100L, /* sec */ \
        0L /* nanosec */ \
    }, \
```

(continues on next page)

(continued from previous page)

```
    5, /* initial_participant_announcements */ \
    {   /*initial_participant_announcement_period */ \
        1L, /* sec */ \
        0L /* nanosec */ \
    }, \
    DDS_BOOLEAN_FALSE, /* cache_serialized_samples */ \
    DDS_LENGTH_AUTO, /* max_participant_locators */ \
    4, /* max_locators_per_discovered_participant */ \
    8, /* max_samples_per_builtin_endpoint_reader */ \
    DDS_LENGTH_UNLIMITED, /* builtin_writer_max_heartbeat_retries */ \
    {   /*builtin_writer_heartbeat_period */ \
        0L, /* sec */ \
        100000000L /* nanosec */ \
    }, \
    1L /* builtin_writer_heartbeats_per_max_samples */ \
    DDS_PARTICIPANT_MESSAGE_READER_RELIABILITY_KIND_INITIALIZER \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=152, columnNumber=35 */
↪
#define RTI_APP_GEN___dpse__HelloWorldAppLibrary_HelloWorldDPSEPubDP_dpse1 \
{ \
    RT_ComponentFactoryProperty_INITIALIZER, /* _parent */ \
    {   /*participant_liveliness_assert_period */ \
        30L, /* sec */ \
        0L /* nanosec */ \
    }, \
    {   /*participant_liveliness_lease_duration */ \
        100L, /* sec */ \
        0L /* nanosec */ \
    }, \
    5, /* initial_participant_announcements */ \
    {   /*initial_participant_announcement_period */ \
        1L, /* sec */ \
        0L /* nanosec */ \
    }, \
    DDS_LENGTH_AUTO, /* max_participant_locators */ \
    4 /* max_locators_per_discovered_participant */ \
    DDS_PARTICIPANT_MESSAGE_READER_RELIABILITY_KIND_INITIALIZER \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=7, columnNumber=38 */
#define RTI_APP_GEN___DPF_QOS_QosLibrary_DefaultProfile \
{ \
    {   /* entity_factory */ \
        DDS_BOOLEAN_FALSE /* autoenable_created_entities */ \
    }, \
    DDS_SYSTEM_RESOURCE_LIMITS_QOS_POLICY_DEFAULT \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=38, columnNumber=67 */
```

(continues on next page)

(continued from previous page)

```
extern const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_initial_peers[2];
extern const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_discovery_enabled_
↪transports[3];
extern const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_transport_enabled_
↪transports[1];
extern const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_user_traffic_enabled_
↪transports[1];
#define RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPDEPubDP \
{ \
    DDS_ENTITY_FACTORY_QOS_POLICY_DEFAULT, \
    {   /* discovery */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪initial_peers, 2, 2), /* initial_peers */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪discovery_enabled_transports, 3, 3), /* enabled_transports */ \
        { \
            { { "dpde1" } }, /* RT_ComponentFactoryId_INITIALIZER */ \
            NDDS_Discovery_Property_INITIALIZER \
        }, /* discovery_component */ \
        DDS_BOOLEAN_FALSE /* accept_unknown_peers */ \
    }, \
    {   /* resource_limits  */ \
        1L, /* local_writer_allocation */ \
        1L, /* local_reader_allocation */ \
        1L, /* local_publisher_allocation */ \
        1L, /* local_subscriber_allocation */ \
        1L, /* local_topic_allocation */ \
        1L, /* local_type_allocation */ \
        8L, /* remote_participant_allocation */ \
        8L, /* remote_writer_allocation */ \
        8L, /* remote_reader_allocation */ \
        32L, /* matching_writer_reader_pair_allocation */ \
        32L, /* matching_reader_writer_pair_allocation */ \
        32L, /* max_receive_ports */ \
        32L, /* max_destination_ports */ \
        65536, /* unbound_data_buffer_size */ \
        500UL /* shmem_ref_transfer_mode_max_segments */ \
    }, \
    DDS_ENTITY_NAME_QOS_POLICY_DEFAULT, \
    DDS_WIRE_PROTOCOL_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪transport_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    {   /* user_traffic */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDEPubDP_user_
↪traffic_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    DDS_TRUST_QOS_POLICY_DEFAULT, \
    DDS_PROPERTY_QOS_POLICY_DEFAULT \
}
```

(continues on next page)

(continued from previous page)

```
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=35, columnNumber=74 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_HelloWorldDPDEPub_
↪HelloWorldDPDEDW_transport_enabled_transports[1];
#define RTI_APP_GEN___DW_QOS_HelloWorldAppLibrary_HelloWorldDPDEPubDP_HelloWorldDPDEPub_
↪HelloWorldDPDEDW \
{ \
    DDS_DEADLINE_QOS_POLICY_DEFAULT, \
    DDS_LIVELINESS_QOS_POLICY_DEFAULT, \
    {   /* history */ \
        DDS_KEEP_LAST_HISTORY_QOS, /* kind */ \
        32L /* depth */ \
    }, \
    {   /* resource_limits */ \
        64L, /* max_samples */ \
        2L, /* max_instances */ \
        32L /* max_samples_per_instance */ \
    }, \
    DDS_OWNERSHIP_QOS_POLICY_DEFAULT, \
    DDS_OWNERSHIP_STRENGTH_QOS_POLICY_DEFAULT, \
    DDS_LATENCY_BUDGET_QOS_POLICY_DEFAULT, \
    {   /* reliability */ \
        DDS_RELIABLE_RELIABILITY_QOS, /* kind */ \
        {   /* max_blocking_time */ \
            0L, /* sec */ \
            100000000L /* nanosec */ \
        } \
    }, \
    DDS_DURABILITY_QOS_POLICY_DEFAULT, \
    DDS_DESTINATION_ORDER_QOS_POLICY_DEFAULT, \
    DDS_TRANSPORT_ENCAPSULATION_QOS_POLICY_DEFAULT, \
    DDS_DATA_REPRESENTATION_QOS_POLICY_DEFAULT, \
    {   /* protocol */ \
        DDS_RTPS_AUTO_ID, /* rtps_object_id */ \
        { /* rtps_reliable_writer */ \
            {   /* heartbeat_period */ \
                0L, /* sec */ \
                250000000L /* nanosec */ \
            },  \
            1L, /* heartbeats_per_max_samples */ \
            DDS_LENGTH_UNLIMITED, /* max_send_window */ \
            DDS_LENGTH_UNLIMITED, /* max_heartbeat_retries */ \
            {   /* first_write_sequence_number */ \
                0, /* high */ \
                1  /* low */ \
            } \
        }, \
        DDS_BOOLEAN_TRUE /* serialize_on_write */ \
    }, \
    DDS_TYPESUPPORT_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
```

(continues on next page)

(continued from previous page)

```
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪HelloWorldDPDEPub_HelloWorldDPDEDW_transport_enabled_transports, 1, 1) /* enabled_
↪transports */ \
    }, \
    RTI_MANAGEMENT_QOS_POLICY_DEFAULT, \
    DDS_DATAWRITERRESOURCE_LIMITS_QOS_POLICY_DEFAULT, \
    DDS_PUBLISH_MODE_QOS_POLICY_DEFAULT, \
    DDS_DATAWRITERQOS_TRUST_INITIALIZER \
    DDS_DATAWRITERQOS_APPGEN_INITIALIZER \
    NULL, \
    DDS_DataWriterTransferModeQosPolicy_INITIALIZER \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=48, columnNumber=67 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_initial_peers[2];
extern const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_discovery_enabled_
↪transports[3];
extern const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_transport_enabled_
↪transports[1];
extern const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_user_traffic_enabled_
↪transports[1];
#define RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPDESubDP \
{ \
    DDS_ENTITY_FACTORY_QOS_POLICY_DEFAULT, \
    {   /* discovery */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪initial_peers, 2, 2), /* initial_peers */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪discovery_enabled_transports, 3, 3), /* enabled_transports */ \
        { \
            { { "dpde1" } }, /* RT_ComponentFactoryId_INITIALIZER */ \
            NDDS_Discovery_Property_INITIALIZER \
        }, /* discovery_component */ \
        DDS_BOOLEAN_FALSE /* accept_unknown_peers */ \
    }, \
    {   /* resource_limits  */ \
        1L, /* local_writer_allocation */ \
        1L, /* local_reader_allocation */ \
        1L, /* local_publisher_allocation */ \
        1L, /* local_subscriber_allocation */ \
        1L, /* local_topic_allocation */ \
        1L, /* local_type_allocation */ \
        8L, /* remote_participant_allocation */ \
        8L, /* remote_writer_allocation */ \
        8L, /* remote_reader_allocation */ \
        32L, /* matching_writer_reader_pair_allocation */ \
        32L, /* matching_reader_writer_pair_allocation */ \
        32L, /* max_receive_ports */ \
        32L, /* max_destination_ports */ \
        65536, /* unbound_data_buffer_size */ \
        500UL /* shmem_ref_transfer_mode_max_segments */ \
```

(continues on next page)

(continued from previous page)

```
    }, \
    DDS_ENTITY_NAME_QOS_POLICY_DEFAULT, \
    DDS_WIRE_PROTOCOL_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪transport_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    {   /* user_traffic */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDESubDP_user_
↪traffic_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    DDS_TRUST_QOS_POLICY_DEFAULT, \
    DDS_PROPERTY_QOS_POLICY_DEFAULT \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=45, columnNumber=74 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_HelloWorldDPDESub_
↪HelloWorldDPDEDR_transport_enabled_transports[2];
#define RTI_APP_GEN___DR_QOS_HelloWorldAppLibrary_HelloWorldDPDESubDP_HelloWorldDPDESub_
↪HelloWorldDPDEDR \
{ \
    DDS_DEADLINE_QOS_POLICY_DEFAULT, \
    DDS_LIVELINESS_QOS_POLICY_DEFAULT, \
    {   /* history */ \
        DDS_KEEP_LAST_HISTORY_QOS, /* kind */ \
        32L /* depth */ \
    }, \
    {   /* resource_limits */ \
        64L, /* max_samples */ \
        2L, /* max_instances */ \
        32L /* max_samples_per_instance */ \
    }, \
    DDS_OWNERSHIP_QOS_POLICY_DEFAULT, \
    DDS_LATENCY_BUDGET_QOS_POLICY_DEFAULT, \
    {   /* reliability */ \
        DDS_RELIABLE_RELIABILITY_QOS, /* kind */ \
        {   /* max_blocking_time */ \
            0L, /* sec */ \
            0L /* nanosec */ \
        } \
    }, \
    DDS_DURABILITY_QOS_POLICY_DEFAULT, \
    DDS_DESTINATION_ORDER_QOS_POLICY_DEFAULT, \
    DDS_TRANSPORT_ENCAPSULATION_QOS_POLICY_DEFAULT, \
    DDS_DATA_REPRESENTATION_QOS_POLICY_DEFAULT, \
    DDS_TYPESUPPORT_QOS_POLICY_DEFAULT, \
    DDS_DATA_READER_PROTOCOL_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪HelloWorldDPDESub_HelloWorldDPDEDR_transport_enabled_transports, 2, 2) /* enabled_
↪transports */ \
```

(continues on next page)

(continued from previous page)

```
    }, \
    {   /* reader_resource_limits */ \
        10L, /* max_remote_writers */ \
        10L, /* max_remote_writers_per_instance */ \
        1L, /* max_samples_per_remote_writer */ \
        1L, /* max_outstanding_reads */ \
        DDS_NO_INSTANCE_REPLACEMENT_QOS, /* instance_replacement */ \
        4L, /* max_routes_per_writer */ \
        DDS_MAX_AUTO, /* max_fragmented_samples */ \
        DDS_MAX_AUTO, /* max_fragmented_samples_per_remote_writer */ \
        DDS_SIZE_AUTO /* shmem_ref_transfer_mode_attached_segment_allocation */ \
    }, \
    RTI_MANAGEMENT_QOS_POLICY_DEFAULT, \
    DDS_DATAREADERQOS_TRUST_INITIALIZER \
    DDS_DATAREADERQOS_APPGEN_INITIALIZER \
    NULL \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=58, columnNumber=67 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_initial_peers[2];
extern const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_discovery_enabled_
↪transports[3];
extern const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_transport_enabled_
↪transports[1];
extern const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_user_traffic_enabled_
↪transports[1];
#define RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPSEPubDP \
{ \
    DDS_ENTITY_FACTORY_QOS_POLICY_DEFAULT, \
    {   /* discovery */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪initial_peers, 2, 2), /* initial_peers */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪discovery_enabled_transports, 3, 3), /* enabled_transports */ \
        { \
            { { "dpse1" } }, /* RT_ComponentFactoryId_INITIALIZER */ \
            NDDS_Discovery_Property_INITIALIZER \
        }, /* discovery_component */ \
        DDS_BOOLEAN_FALSE /* accept_unknown_peers */ \
    }, \
    {   /* resource_limits  */ \
        1L, /* local_writer_allocation */ \
        1L, /* local_reader_allocation */ \
        1L, /* local_publisher_allocation */ \
        1L, /* local_subscriber_allocation */ \
        1L, /* local_topic_allocation */ \
        1L, /* local_type_allocation */ \
        8L, /* remote_participant_allocation */ \
        8L, /* remote_writer_allocation */ \
        8L, /* remote_reader_allocation */ \
        32L, /* matching_writer_reader_pair_allocation */ \
```

(continues on next page)

(continued from previous page)

```
        32L, /* matching_reader_writer_pair_allocation */ \
        32L, /* max_receive_ports */ \
        32L, /* max_destination_ports */ \
        65536, /* unbound_data_buffer_size */ \
        500UL /* shmem_ref_transfer_mode_max_segments */ \
    }, \
    DDS_ENTITY_NAME_QOS_POLICY_DEFAULT, \
    DDS_WIRE_PROTOCOL_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪transport_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    {   /* user_traffic */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSEPubDP_user_
↪traffic_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    DDS_TRUST_QOS_POLICY_DEFAULT, \
    DDS_PROPERTY_QOS_POLICY_DEFAULT \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=55, columnNumber=74 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_HelloWorldDPSEPub_
↪HelloWorldDPSEDW_transport_enabled_transports[1];
#define RTI_APP_GEN___DW_QOS_HelloWorldAppLibrary_HelloWorldDPSEPubDP_HelloWorldDPSEPub_
↪HelloWorldDPSEDW \
{ \
    DDS_DEADLINE_QOS_POLICY_DEFAULT, \
    DDS_LIVELINESS_QOS_POLICY_DEFAULT, \
    {   /* history */ \
        DDS_KEEP_LAST_HISTORY_QOS, /* kind */ \
        32L /* depth */ \
    }, \
    {   /* resource_limits */ \
        64L, /* max_samples */ \
        2L, /* max_instances */ \
        32L /* max_samples_per_instance */ \
    }, \
    DDS_OWNERSHIP_QOS_POLICY_DEFAULT, \
    DDS_OWNERSHIP_STRENGTH_QOS_POLICY_DEFAULT, \
    DDS_LATENCY_BUDGET_QOS_POLICY_DEFAULT, \
    {   /* reliability */ \
        DDS_RELIABLE_RELIABILITY_QOS, /* kind */ \
        {   /* max_blocking_time */ \
            0L, /* sec */ \
            100000000L /* nanosec */ \
        } \
    }, \
    DDS_DURABILITY_QOS_POLICY_DEFAULT, \
    DDS_DESTINATION_ORDER_QOS_POLICY_DEFAULT, \
    DDS_TRANSPORT_ENCAPSULATION_QOS_POLICY_DEFAULT, \
    DDS_DATA_REPRESENTATION_QOS_POLICY_DEFAULT, \
```

(continues on next page)

(continued from previous page)

```
    {   /* protocol */ \
        1UL, /* rtps_object_id */ \
        { /* rtps_reliable_writer */ \
            {   /* heartbeat_period */ \
                0L, /* sec */ \
                250000000L /* nanosec */ \
            },  \
            1L, /* heartbeats_per_max_samples */ \
            DDS_LENGTH_UNLIMITED, /* max_send_window */ \
            DDS_LENGTH_UNLIMITED, /* max_heartbeat_retries */ \
            {   /* first_write_sequence_number */ \
                0, /* high */ \
                1  /* low */ \
            } \
        }, \
        DDS_BOOLEAN_TRUE /* serialize_on_write */ \
    }, \
    DDS_TYPESUPPORT_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪HelloWorldDPSEPub_HelloWorldDPSEDW_transport_enabled_transports, 1, 1) /* enabled_
↪transports */ \
    }, \
    RTI_MANAGEMENT_QOS_POLICY_DEFAULT, \
    DDS_DATAWRITERRESOURCE_LIMITS_QOS_POLICY_DEFAULT, \
    DDS_PUBLISH_MODE_QOS_POLICY_DEFAULT, \
    DDS_DATAWRITERQOS_TRUST_INITIALIZER \
    DDS_DATAWRITERQOS_APPGEN_INITIALIZER \
    NULL, \
    DDS_DataWriterTransferModeQosPolicy_INITIALIZER \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=68, columnNumber=67 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_initial_peers[2];
extern const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_discovery_enabled_
↪transports[3];
extern const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_transport_enabled_
↪transports[1];
extern const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_user_traffic_enabled_
↪transports[1];
#define RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPSESubDP \
{ \
    DDS_ENTITY_FACTORY_QOS_POLICY_DEFAULT, \
    {   /* discovery */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪initial_peers, 2, 2), /* initial_peers */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪discovery_enabled_transports, 3, 3), /* enabled_transports */ \
        { \
            { { "dpse1" } }, /* RT_ComponentFactoryId_INITIALIZER */ \
            NDDS_Discovery_Property_INITIALIZER \
```

(continues on next page)

(continued from previous page)

```
        }, /* discovery_component */ \
        DDS_BOOLEAN_FALSE /* accept_unknown_peers */ \
    }, \
    {   /* resource_limits  */ \
        1L, /* local_writer_allocation */ \
        1L, /* local_reader_allocation */ \
        1L, /* local_publisher_allocation */ \
        1L, /* local_subscriber_allocation */ \
        1L, /* local_topic_allocation */ \
        1L, /* local_type_allocation */ \
        8L, /* remote_participant_allocation */ \
        8L, /* remote_writer_allocation */ \
        8L, /* remote_reader_allocation */ \
        32L, /* matching_writer_reader_pair_allocation */ \
        32L, /* matching_reader_writer_pair_allocation */ \
        32L, /* max_receive_ports */ \
        32L, /* max_destination_ports */ \
        65536, /* unbound_data_buffer_size */ \
        500UL /* shmem_ref_transfer_mode_max_segments */ \
    }, \
    DDS_ENTITY_NAME_QOS_POLICY_DEFAULT, \
    DDS_WIRE_PROTOCOL_QOS_POLICY_DEFAULT, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪transport_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    {   /* user_traffic */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSESubDP_user_
↪traffic_enabled_transports, 1, 1) /* enabled_transports */ \
    }, \
    DDS_TRUST_QOS_POLICY_DEFAULT, \
    DDS_PROPERTY_QOS_POLICY_DEFAULT \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=65, columnNumber=74 */
extern const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_HelloWorldDPSESub_
↪HelloWorldDPSEDR_transport_enabled_transports[2];
#define RTI_APP_GEN___DR_QOS_HelloWorldAppLibrary_HelloWorldDPSESubDP_HelloWorldDPSESub_
↪HelloWorldDPSEDR \
{ \
    DDS_DEADLINE_QOS_POLICY_DEFAULT, \
    DDS_LIVELINESS_QOS_POLICY_DEFAULT, \
    {   /* history */ \
        DDS_KEEP_LAST_HISTORY_QOS, /* kind */ \
        32L /* depth */ \
    }, \
    {   /* resource_limits */ \
        64L, /* max_samples */ \
        2L, /* max_instances */ \
        32L /* max_samples_per_instance */ \
    }, \
```

(continues on next page)

(continued from previous page)

```
    DDS_OWNERSHIP_QOS_POLICY_DEFAULT, \
    DDS_LATENCY_BUDGET_QOS_POLICY_DEFAULT, \
    {   /* reliability */ \
        DDS_RELIABLE_RELIABILITY_QOS, /* kind */ \
        {   /* max_blocking_time */ \
            0L, /* sec */ \
            0L /* nanosec */ \
        } \
    }, \
    DDS_DURABILITY_QOS_POLICY_DEFAULT, \
    DDS_DESTINATION_ORDER_QOS_POLICY_DEFAULT, \
    DDS_TRANSPORT_ENCAPSULATION_QOS_POLICY_DEFAULT, \
    DDS_DATA_REPRESENTATION_QOS_POLICY_DEFAULT, \
    DDS_TYPESUPPORT_QOS_POLICY_DEFAULT, \
    {   /* protocol */ \
        2UL /* rtps_object_id */ \
    }, \
    {   /* transports */ \
        REDA_StringSeq_INITIALIZER_W_LOAN(HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪HelloWorldDPSESub_HelloWorldDPSEDR_transport_enabled_transports, 2, 2) /* enabled_
↪transports */ \
    }, \
    {   /* reader_resource_limits */ \
        10L, /* max_remote_writers */ \
        10L, /* max_remote_writers_per_instance */ \
        1L, /* max_samples_per_remote_writer */ \
        1L, /* max_outstanding_reads */ \
        DDS_NO_INSTANCE_REPLACEMENT_QOS, /* instance_replacement */ \
        4L, /* max_routes_per_writer */ \
        DDS_MAX_AUTO, /* max_fragmented_samples */ \
        DDS_MAX_AUTO, /* max_fragmented_samples_per_remote_writer */ \
        DDS_SIZE_AUTO /* shmem_ref_transfer_mode_attached_segment_allocation */ \
    }, \
    RTI_MANAGEMENT_QOS_POLICY_DEFAULT, \
    DDS_DATAREADERQOS_TRUST_INITIALIZER \
    DDS_DATAREADERQOS_APPGEN_INITIALIZER \
    NULL \
}
extern struct DPDE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪dpde[1];
extern struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪udpv4[1];
extern const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_
↪HelloWorldDPDEPubDP_unregister_components[2];
extern const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_
↪HelloWorldDPDEPubDP_register_components[2];
#define RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPDEPubDP \
{ \
    2UL, /* unregister_count */ \
    HelloWorldAppLibrary_HelloWorldDPDEPubDP_unregister_components, /* unregister_
↪components */ \
```

(continues on next page)

(continued from previous page)

```
    2UL, /* register_count */ \
    HelloWorldAppLibrary_HelloWorldDPDEPubDP_register_components, /* register_components␣
↪*/ \
    RTI_APP_GEN___DPF_QOS_QosLibrary_DefaultProfile /* factory_qos */ \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=32, columnNumber=62 */
extern const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_
↪HelloWorldDPDEPubDP_type_registrations[1];
extern const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_topics[1];
extern const struct APPGEN_PublisherModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪publishers[1];
#define RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPDEPubDP \
{ \
    "HelloWorldDPDEPubDP", /* name */ \
    RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPDEPubDP, /* domain_participant_
↪factory */ \
    RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPDEPubDP, /* participant_qos */␣
↪\
    0L, /* domain_id */ \
    1UL, /* type_registration_count */ \
    HelloWorldAppLibrary_HelloWorldDPDEPubDP_type_registrations, /* type_registrations */
↪ \
    1UL, /* topic_count */ \
    HelloWorldAppLibrary_HelloWorldDPDEPubDP_topics, /* topics */ \
    1UL, /* publisher_count */ \
    HelloWorldAppLibrary_HelloWorldDPDEPubDP_publishers, /* publishers */ \
    0UL, /* subscriber_count */ \
    NULL, /* subscribers */ \
    0UL, /* remote_participant_count */ \
    NULL, /* remote_participants */ \
    0UL, /* flow_controller_count */ \
    NULL, /* flow_controllers */ \
}
extern struct DPDE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪dpde[1];
extern struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪udpv4[1];
extern const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_
↪HelloWorldDPDESubDP_unregister_components[2];
extern const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_
↪HelloWorldDPDESubDP_register_components[2];
#define RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPDESubDP \
{ \
    2UL, /* unregister_count */ \
    HelloWorldAppLibrary_HelloWorldDPDESubDP_unregister_components, /* unregister_
↪components */ \
    2UL, /* register_count */ \
    HelloWorldAppLibrary_HelloWorldDPDESubDP_register_components, /* register_components␣
↪*/ \
    RTI_APP_GEN___DPF_QOS_QosLibrary_DefaultProfile /* factory_qos */ \
```

(continues on next page)

(continued from previous page)

```
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=42, columnNumber=62 */
extern const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_
↪HelloWorldDPDESubDP_type_registrations[1];
extern const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPDESubDP_topics[1];
extern const struct APPGEN_SubscriberModel HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪subscribers[1];
#define RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPDESubDP \
{ \
    "HelloWorldDPDESubDP", /* name */ \
    RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPDESubDP, /* domain_participant_
↪factory */ \
    RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPDESubDP, /* participant_qos */␣
↪\
    0L, /* domain_id */ \
    1UL, /* type_registration_count */ \
    HelloWorldAppLibrary_HelloWorldDPDESubDP_type_registrations, /* type_registrations */
↪ \
    1UL, /* topic_count */ \
    HelloWorldAppLibrary_HelloWorldDPDESubDP_topics, /* topics */ \
    0UL, /* publisher_count */ \
    NULL, /* publishers */ \
    1UL, /* subscriber_count */ \
    HelloWorldAppLibrary_HelloWorldDPDESubDP_subscribers, /* subscribers */ \
    0UL, /* remote_participant_count */ \
    NULL, /* remote_participants */ \
    0UL, /* flow_controller_count */ \
    NULL, /* flow_controllers */ \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=64, columnNumber=82 */
#define RTI_APP_GEN__RSD_HelloWorldAppLibrary_HelloWorldDPSEPubDP_HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_HelloWorldDPSESub_HelloWorldDPSEDR \
{ \
    { /* subscription_data */ \
        { \
            { 0, 0, 0, 2 } /* key */ \
        }, \
        { \
            { 0, 0, 0, 0 } /* participant_key */ \
        }, \
        "HelloWorldTopic", /* topic_name */ \
        "HelloWorldType", /* type_name */ \
        DDS_DEADLINE_QOS_POLICY_DEFAULT, \
        DDS_OWNERSHIP_QOS_POLICY_DEFAULT, \
        DDS_LATENCY_BUDGET_QOS_POLICY_DEFAULT, \
        {   /* reliability */ \
            DDS_RELIABLE_RELIABILITY_QOS, /* kind */ \
            {   /* max_blocking_time */ \
                0L, /* sec */ \
```

(continues on next page)

(continued from previous page)

```
                0L /* nanosec */ \
            } \
        }, \
        DDS_LIVELINESS_QOS_POLICY_DEFAULT, \
        DDS_DURABILITY_QOS_POLICY_DEFAULT, \
        DDS_DESTINATION_ORDER_QOS_POLICY_DEFAULT, \
        DDS_SEQUENCE_INITIALIZER, \
        DDS_SEQUENCE_INITIALIZER, \
        DDS_DATA_REPRESENTATION_QOS_POLICY_DEFAULT \
        DDS_TRUST_SUBSCRIPTION_DATA_INITIALIZER \
    }, \
    HelloWorldTypePlugin_get /* get_type_plugin */ \
}
extern struct DPSE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪dpse[1];
extern struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪udpv4[1];
extern const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_unregister_components[2];
extern const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_register_components[2];
#define RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPSEPubDP \
{ \
    2UL, /* unregister_count */ \
    HelloWorldAppLibrary_HelloWorldDPSEPubDP_unregister_components, /* unregister_
↪components */ \
    2UL, /* register_count */ \
    HelloWorldAppLibrary_HelloWorldDPSEPubDP_register_components, /* register_components␣
↪*/ \
    RTI_APP_GEN___DPF_QOS_QosLibrary_DefaultProfile /* factory_qos */ \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=52, columnNumber=62 */
extern const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_type_registrations[1];
extern const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_topics[1];
extern const struct APPGEN_PublisherModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪publishers[1];
extern const struct APPGEN_RemoteSubscriptionModel HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_remote_subscribers[1];
extern const struct APPGEN_RemoteParticipantModel HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_remote_participants[1];
#define RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPSEPubDP \
{ \
    "HelloWorldDPSEPubDP", /* name */ \
    RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPSEPubDP, /* domain_participant_
↪factory */ \
    RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPSEPubDP, /* participant_qos */␣
↪\
    0L, /* domain_id */ \
    1UL, /* type_registration_count */ \
```

(continues on next page)

(continued from previous page)

```
    HelloWorldAppLibrary_HelloWorldDPSEPubDP_type_registrations, /* type_registrations */
↪ \
    1UL, /* topic_count */ \
    HelloWorldAppLibrary_HelloWorldDPSEPubDP_topics, /* topics */ \
    1UL, /* publisher_count */ \
    HelloWorldAppLibrary_HelloWorldDPSEPubDP_publishers, /* publishers */ \
    0UL, /* subscriber_count */ \
    NULL, /* subscribers */ \
    1UL, /* remote_participant_count */ \
    HelloWorldAppLibrary_HelloWorldDPSEPubDP_remote_participants, /* remote_participants␣
↪*/ \
    0UL, /* flow_controller_count */ \
    NULL, /* flow_controllers */ \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=54, columnNumber=82 */
#define RTI_APP_GEN__RPD_HelloWorldAppLibrary_HelloWorldDPSESubDP_HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_HelloWorldDPSEPub_HelloWorldDPSEDW \
{ \
    { /* publication_data */ \
        { \
            { 0, 0, 0, 1 } /* key */ \
        }, \
        { \
            { 0, 0, 0, 0 } /* participant_key */ \
        }, \
        "HelloWorldTopic", /* topic_name */ \
        "HelloWorldType", /* type_name */ \
        DDS_DEADLINE_QOS_POLICY_DEFAULT, \
        DDS_OWNERSHIP_QOS_POLICY_DEFAULT, \
        DDS_OWNERSHIP_STRENGTH_QOS_POLICY_DEFAULT, \
        DDS_LATENCY_BUDGET_QOS_POLICY_DEFAULT, \
        {   /* reliability */ \
            DDS_RELIABLE_RELIABILITY_QOS, /* kind */ \
            {   /* max_blocking_time */ \
                0L, /* sec */ \
                100000000L /* nanosec */ \
            } \
        }, \
        DDS_LIVELINESS_QOS_POLICY_DEFAULT, \
        DDS_DURABILITY_QOS_POLICY_DEFAULT, \
        DDS_DESTINATION_ORDER_QOS_POLICY_DEFAULT, \
        DDS_SEQUENCE_INITIALIZER, \
        DDS_DATA_REPRESENTATION_QOS_POLICY_DEFAULT \
        DDS_TRUST_PUBLICATION_DATA_INITIALIZER \
    }, \
    HelloWorldTypePlugin_get /* get_type_plugin */ \
}
extern struct DPSE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪dpse[1];
extern struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪udpv4[1];
```

(continues on next page)

(continued from previous page)

```
extern const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_unregister_components[2];
extern const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_register_components[2];
#define RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPSESubDP \
{ \
    2UL, /* unregister_count */ \
    HelloWorldAppLibrary_HelloWorldDPSESubDP_unregister_components, /* unregister_
↪components */ \
    2UL, /* register_count */ \
    HelloWorldAppLibrary_HelloWorldDPSESubDP_register_components, /* register_components␣
↪*/ \
    RTI_APP_GEN___DPF_QOS_QosLibrary_DefaultProfile /* factory_qos */ \
}
/* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=62, columnNumber=62 */
extern const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_type_registrations[1];
extern const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPSESubDP_topics[1];
extern const struct APPGEN_SubscriberModel HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪subscribers[1];
extern const struct APPGEN_RemotePublicationModel HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_remote_publishers[1];
extern const struct APPGEN_RemoteParticipantModel HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_remote_participants[1];
#define RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPSESubDP \
{ \
    "HelloWorldDPSESubDP", /* name */ \
    RTI_APP_GEN__DPF_HelloWorldAppLibrary_HelloWorldDPSESubDP, /* domain_participant_
↪factory */ \
    RTI_APP_GEN___DP_QOS_HelloWorldAppLibrary_HelloWorldDPSESubDP, /* participant_qos */␣
↪\
    0L, /* domain_id */ \
    1UL, /* type_registration_count */ \
    HelloWorldAppLibrary_HelloWorldDPSESubDP_type_registrations, /* type_registrations */
↪ \
    1UL, /* topic_count */ \
    HelloWorldAppLibrary_HelloWorldDPSESubDP_topics, /* topics */ \
    0UL, /* publisher_count */ \
    NULL, /* publishers */ \
    1UL, /* subscriber_count */ \
    HelloWorldAppLibrary_HelloWorldDPSESubDP_subscribers, /* subscribers */ \
    1UL, /* remote_participant_count */ \
    HelloWorldAppLibrary_HelloWorldDPSESubDP_remote_participants, /* remote_participants␣
↪*/ \
    0UL, /* flow_controller_count */ \
    NULL, /* flow_controllers */ \
}
extern const struct APPGEN_DomainParticipantModel HelloWorldAppLibrary_participants[4];
#define RTI_APP_GEN__LIB_HelloWorldAppLibrary \
{ \
```

(continues on next page)

(continued from previous page)

```
    "HelloWorldAppLibrary", /* library_name */ \
    4UL, /* participant_count */ \
    HelloWorldAppLibrary_participants /* participants */ \
}
extern const struct APPGEN_LibraryModel HelloWorld_libraries[1];
```

Example generated source configuration file HelloWorldAppgen.c:

```
/*
WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.
This file was generated from HelloWorld.xml using "rtiddsmag."
The rtiddsmag tool is part of the RTI Connext distribution.
For more information, type 'rtiddsmag -help' at a command shell
or consult the RTI Connext manual.
*/
#include "HelloWorldAppgen.h"
const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_initial_peers[2] =
{
    "127.0.0.1",
    "239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_discovery_enabled_
↪transports[3] =
{
    "udp1://",
    "udp1://127.0.0.1",
    "udp1://239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_transport_enabled_
↪transports[1] =
{
    "udp1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_user_traffic_enabled_
↪transports[1] =
{
    "udp1://"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDEPubDP_HelloWorldDPDEPub_
↪HelloWorldDPDEDW_transport_enabled_transports[1] =
{
    "udp1://"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_initial_peers[2] =
{
    "127.0.0.1",
    "239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_discovery_enabled_
↪transports[3] =
{
```

(continues on next page)

(continued from previous page)

```
    "udp1://",
    "udp1://127.0.0.1",
    "udp1://239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_transport_enabled_
↪transports[1] =
{
    "udp1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_user_traffic_enabled_
↪transports[1] =
{
    "udp1://"
};
const char *const HelloWorldAppLibrary_HelloWorldDPDESubDP_HelloWorldDPDESub_
↪HelloWorldDPDEDR_transport_enabled_transports[2] =
{
    "udp1://",
    "udp1://127.0.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_initial_peers[2] =
{
    "127.0.0.1",
    "239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_discovery_enabled_
↪transports[3] =
{
    "udp1://",
    "udp1://127.0.0.1",
    "udp1://239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_transport_enabled_
↪transports[1] =
{
    "udp1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_user_traffic_enabled_
↪transports[1] =
{
    "udp1://"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSEPubDP_HelloWorldDPSEPub_
↪HelloWorldDPSEDW_transport_enabled_transports[1] =
{
    "udp1://"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_initial_peers[2] =
{
    "127.0.0.1",
    "239.255.0.1"
```

(continues on next page)

(continued from previous page)

```
};
const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_discovery_enabled_
↪transports[3] =
{
    "udp1://",
    "udp1://127.0.0.1",
    "udp1://239.255.0.1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_transport_enabled_
↪transports[1] =
{
    "udp1"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_user_traffic_enabled_
↪transports[1] =
{
    "udp1://"
};
const char *const HelloWorldAppLibrary_HelloWorldDPSESubDP_HelloWorldDPSESub_
↪HelloWorldDPSEDR_transport_enabled_transports[2] =
{
    "udp1://",
    "udp1://127.0.0.1"
};
const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪unregister_components[2] =
{
    {
        "_udp", /* NETIO_DEFAULT_UDP_NAME */
        NULL, /* udp struct RT_ComponentFactoryProperty** */
        NULL  /* udp struct RT_ComponentFactoryListener** */
    },
    {
        "_intra", /* NETIO_DEFAULT_INTRA_NAME */
        NULL, /* _intra struct RT_ComponentFactoryProperty** */
        NULL  /* _intra struct RT_ComponentFactoryListener** */
    }
};
struct DPDE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPDEPubDP_dpde[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=145, columnNumber=35 */
↪
    RTI_APP_GEN___dpde__HelloWorldAppLibrary_HelloWorldDPDEPubDP_dpde1
};
struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPDEPubDP_udpv4[1] =
{
    RTI_APP_GEN___udpv4__HelloWorldAppLibrary_HelloWorldDPDEPubDP_udp1
};
const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪register_components[2] =
```

(continues on next page)

(continued from previous page)

```
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=145, columnNumber=35 */
↪
    {
        "dpde1", /* register_name */
        DPDE_DiscoveryFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPDEPubDP_dpde[0]._parent, /* register_property␣
↪*/
        NULL /* register_listener */
    },
    {
        "udp1", /* register_name */
        UDP_InterfaceFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPDEPubDP_udpv4[0]._parent._parent, /* register_
↪property */
        NULL /* register_listener */
    }
};
const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_type_
↪registrations[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=20, columnNumber=72 */
    {
        "HelloWorldType", /* registered_type_name */
        HelloWorldTypePlugin_get /* get_type_plugin */
    }
};
const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_topics[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=23, columnNumber=78 */
    {
        "HelloWorldTopic", /* topic_name */
        "HelloWorldType", /* type_name */
        DDS_TopicQos_INITIALIZER /* topic_qos*/
    }
};
const struct APPGEN_DataWriterModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_publisher_
↪HelloWorldDPDEPub_data_writers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=34, columnNumber=82 */
    {
        "HelloWorldDPDEDW", /* name */
        1UL, /* multiplicity */
        "HelloWorldTopic", /* topic_name */
        RTI_APP_GEN___DW_QOS_HelloWorldAppLibrary_HelloWorldDPDEPubDP_HelloWorldDPDEPub_
↪HelloWorldDPDEDW /* writer_qos */
    }
```

(continues on next page)

(continued from previous page)

```
};
const struct APPGEN_PublisherModel HelloWorldAppLibrary_HelloWorldDPDEPubDP_
↪publishers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=33, columnNumber=49 */
    {
        "HelloWorldDPDEPub", /* name */
        1UL, /* multiplicity */
        DDS_PublisherQos_INITIALIZER, /* publisher_qos */
        1UL, /* writer_count */
        HelloWorldAppLibrary_HelloWorldDPDEPubDP_publisher_HelloWorldDPDEPub_data_
↪writers /* data_writers */
    }
};
const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪unregister_components[2] =
{
    {
        "_udp", /* NETIO_DEFAULT_UDP_NAME */
        NULL, /* udp struct RT_ComponentFactoryProperty** */
        NULL  /* udp struct RT_ComponentFactoryListener** */
    },
    {
        "_intra", /* NETIO_DEFAULT_INTRA_NAME */
        NULL, /* _intra struct RT_ComponentFactoryProperty** */
        NULL  /* _intra struct RT_ComponentFactoryListener** */
    }
};
struct DPDE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPDESubDP_dpde[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=145, columnNumber=35 */
↪
    RTI_APP_GEN___dpde__HelloWorldAppLibrary_HelloWorldDPDEPubDP_dpde1
};
struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPDESubDP_udpv4[1] =
{
    RTI_APP_GEN___udpv4__HelloWorldAppLibrary_HelloWorldDPDEPubDP_udp1
};
const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪register_components[2] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=145, columnNumber=35 */
↪
    {
        "dpde1", /* register_name */
        DPDE_DiscoveryFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPDESubDP_dpde[0]._parent, /* register_property␣
↪*/
```

(continues on next page)

(continued from previous page)

```
        NULL /* register_listener */
    },
    {
        "udp1", /* register_name */
        UDP_InterfaceFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPDESubDP_udpv4[0]._parent._parent, /* register_
↪property */
        NULL /* register_listener */
    }
};
const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_HelloWorldDPDESubDP_type_
↪registrations[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=20, columnNumber=72 */
    {
        "HelloWorldType", /* registered_type_name */
        HelloWorldTypePlugin_get /* get_type_plugin */
    }
};
const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPDESubDP_topics[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=23, columnNumber=78 */
    {
        "HelloWorldTopic", /* topic_name */
        "HelloWorldType", /* type_name */
        DDS_TopicQos_INITIALIZER /* topic_qos*/
    }
};
const struct APPGEN_DataReaderModel HelloWorldAppLibrary_HelloWorldDPDESubDP_subscriber_
↪HelloWorldDPDESub_data_readers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=44, columnNumber=82 */
    {
        "HelloWorldDPDEDR", /* name */
        1UL, /* multiplicity */
        "HelloWorldTopic", /* topic_name */
        RTI_APP_GEN___DR_QOS_HelloWorldAppLibrary_HelloWorldDPDESubDP_HelloWorldDPDESub_
↪HelloWorldDPDEDR /* reader_qos */
    }
};
const struct APPGEN_SubscriberModel HelloWorldAppLibrary_HelloWorldDPDESubDP_
↪subscribers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=43, columnNumber=50 */
    {
        "HelloWorldDPDESub", /* name */
        1UL, /* multiplicity */
```

(continues on next page)

(continued from previous page)

```
        DDS_SubscriberQos_INITIALIZER, /* subscriber_qos */
        1UL, /* reader_count */
        HelloWorldAppLibrary_HelloWorldDPDESubDP_subscriber_HelloWorldDPDESub_data_
↪readers /* data_readers */
    }
};
const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪unregister_components[2] =
{
    {
        "_udp", /* NETIO_DEFAULT_UDP_NAME */
        NULL, /* udp struct RT_ComponentFactoryProperty** */
        NULL  /* udp struct RT_ComponentFactoryListener** */
    },
    {
        "_intra", /* NETIO_DEFAULT_INTRA_NAME */
        NULL, /* _intra struct RT_ComponentFactoryProperty** */
        NULL  /* _intra struct RT_ComponentFactoryListener** */
    }
};
struct DPSE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPSEPubDP_dpse[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=152, columnNumber=35 */
↪
    RTI_APP_GEN___dpse__HelloWorldAppLibrary_HelloWorldDPSEPubDP_dpse1
};
struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPSEPubDP_udpv4[1] =
{
    RTI_APP_GEN___udpv4__HelloWorldAppLibrary_HelloWorldDPDEPubDP_udp1
};
const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪register_components[2] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=152, columnNumber=35 */
↪
    {
        "dpse1", /* register_name */
        DPSE_DiscoveryFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPSEPubDP_dpse[0]._parent, /* register_property␣
↪*/
        NULL /* register_listener */
    },
    {
        "udp1", /* register_name */
        UDP_InterfaceFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPSEPubDP_udpv4[0]._parent._parent, /* register_
↪property */
        NULL /* register_listener */
    }
```

(continues on next page)

(continued from previous page)

```
};
const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_type_
↪registrations[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=20, columnNumber=72 */
    {
        "HelloWorldType", /* registered_type_name */
        HelloWorldTypePlugin_get /* get_type_plugin */
    }
};
const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_topics[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=23, columnNumber=78 */
    {
        "HelloWorldTopic", /* topic_name */
        "HelloWorldType", /* type_name */
        DDS_TopicQos_INITIALIZER /* topic_qos*/
    }
};
const struct APPGEN_DataWriterModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_publisher_
↪HelloWorldDPSEPub_data_writers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=54, columnNumber=82 */
    {
        "HelloWorldDPSEDW", /* name */
        1UL, /* multiplicity */
        "HelloWorldTopic", /* topic_name */
        RTI_APP_GEN___DW_QOS_HelloWorldAppLibrary_HelloWorldDPSEPubDP_HelloWorldDPSEPub_
↪HelloWorldDPSEDW /* writer_qos */
    }
};
const struct APPGEN_PublisherModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪publishers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=53, columnNumber=49 */
    {
        "HelloWorldDPSEPub", /* name */
        1UL, /* multiplicity */
        DDS_PublisherQos_INITIALIZER, /* publisher_qos */
        1UL, /* writer_count */
        HelloWorldAppLibrary_HelloWorldDPSEPubDP_publisher_HelloWorldDPSEPub_data_
↪writers /* data_writers */
    }
};
const struct APPGEN_RemoteSubscriptionModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪remote_subscribers[1] =
{
```

(continues on next page)

(continued from previous page)

```
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=64, columnNumber=82 */
    RTI_APP_GEN__RSD_HelloWorldAppLibrary_HelloWorldDPSEPubDP_HelloWorldAppLibrary_
↪HelloWorldDPSESubDP_HelloWorldDPSESub_HelloWorldDPSEDR
};
const struct APPGEN_RemoteParticipantModel HelloWorldAppLibrary_HelloWorldDPSEPubDP_
↪remote_participants[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=62, columnNumber=62 */
    {
        "HelloWorldDPSESubDP", /* name */
        0UL, /* remote_publisher_count */
        NULL, /* remote_publishers */
        1UL, /* remote_subscriber_count */
        HelloWorldAppLibrary_HelloWorldDPSEPubDP_remote_subscribers /* remote_
↪subscribers */
    }
};
const struct ComponentFactoryUnregisterModel HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪unregister_components[2] =
{
    {
        "_udp", /* NETIO_DEFAULT_UDP_NAME */
        NULL, /* udp struct RT_ComponentFactoryProperty** */
        NULL  /* udp struct RT_ComponentFactoryListener** */
    },
    {
        "_intra", /* NETIO_DEFAULT_INTRA_NAME */
        NULL, /* _intra struct RT_ComponentFactoryProperty** */
        NULL  /* _intra struct RT_ComponentFactoryListener** */
    }
};
struct DPSE_DiscoveryPluginProperty HelloWorldAppLibrary_HelloWorldDPSESubDP_dpse[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=152, columnNumber=35 */
↪
    RTI_APP_GEN___dpse__HelloWorldAppLibrary_HelloWorldDPSEPubDP_dpse1
};
struct UDP_InterfaceFactoryProperty HelloWorldAppLibrary_HelloWorldDPSESubDP_udpv4[1] =
{
    RTI_APP_GEN___udpv4__HelloWorldAppLibrary_HelloWorldDPDEPubDP_udp1
};
const struct ComponentFactoryRegisterModel HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪register_components[2] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorldQos.xml, lineNumber=152, columnNumber=35 */
↪
    {
```

(continues on next page)

(continued from previous page)

```
        "dpse1", /* register_name */
        DPSE_DiscoveryFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPSESubDP_dpse[0]._parent, /* register_property␣
↪*/
        NULL /* register_listener */
    },
    {
        "udp1", /* register_name */
        UDP_InterfaceFactory_get_interface, /* register_intf */
        &HelloWorldAppLibrary_HelloWorldDPSESubDP_udpv4[0]._parent._parent, /* register_
↪property */
        NULL /* register_listener */
    }
};
const struct APPGEN_TypeRegistrationModel HelloWorldAppLibrary_HelloWorldDPSESubDP_type_
↪registrations[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=20, columnNumber=72 */
    {
        "HelloWorldType", /* registered_type_name */
        HelloWorldTypePlugin_get /* get_type_plugin */
    }
};
const struct APPGEN_TopicModel HelloWorldAppLibrary_HelloWorldDPSESubDP_topics[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=23, columnNumber=78 */
    {
        "HelloWorldTopic", /* topic_name */
        "HelloWorldType", /* type_name */
        DDS_TopicQos_INITIALIZER /* topic_qos*/
    }
};
const struct APPGEN_DataReaderModel HelloWorldAppLibrary_HelloWorldDPSESubDP_subscriber_
↪HelloWorldDPSESub_data_readers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=64, columnNumber=82 */
    {
        "HelloWorldDPSEDR", /* name */
        1UL, /* multiplicity */
        "HelloWorldTopic", /* topic_name */
        RTI_APP_GEN___DR_QOS_HelloWorldAppLibrary_HelloWorldDPSESubDP_HelloWorldDPSESub_
↪HelloWorldDPSEDR /* reader_qos */
    }
};
const struct APPGEN_SubscriberModel HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪subscribers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=63, columnNumber=50 */
```

(continues on next page)

(continued from previous page)

```
    {
        "HelloWorldDPSESub", /* name */
        1UL, /* multiplicity */
        DDS_SubscriberQos_INITIALIZER, /* subscriber_qos */
        1UL, /* reader_count */
        HelloWorldAppLibrary_HelloWorldDPSESubDP_subscriber_HelloWorldDPSESub_data_
↪readers /* data_readers */
    }
};
const struct APPGEN_RemotePublicationModel HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪remote_publishers[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=54, columnNumber=82 */
    RTI_APP_GEN__RPD_HelloWorldAppLibrary_HelloWorldDPSESubDP_HelloWorldAppLibrary_
↪HelloWorldDPSEPubDP_HelloWorldDPSEPub_HelloWorldDPSEDW
};
const struct APPGEN_RemoteParticipantModel HelloWorldAppLibrary_HelloWorldDPSESubDP_
↪remote_participants[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=52, columnNumber=62 */
    {
        "HelloWorldDPSEPubDP", /* name */
        1UL, /* remote_publisher_count */
        HelloWorldAppLibrary_HelloWorldDPSESubDP_remote_publishers, /* remote_publishers␣
↪*/
        0UL, /* remote_subscriber_count */
        NULL /* remote_subscribers */
    }
};
const struct APPGEN_DomainParticipantModel HelloWorldAppLibrary_participants[4] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=32, columnNumber=62 */
    RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPDEPubDP,
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=42, columnNumber=62 */
    RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPDESubDP,
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=52, columnNumber=62 */
    RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPSEPubDP,
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=62, columnNumber=62 */
    RTI_APP_GEN__DP_HelloWorldAppLibrary_HelloWorldDPSESubDP
};
const struct APPGEN_LibraryModel HelloWorld_libraries[1] =
{
    /* XML Source Location: file=c:\shared\connextmicro\rti\ndds_lite\rti_me.2.
↪0\example\C\HelloWorld_appgen\HelloWorld.xml, lineNumber=30, columnNumber=61 */
    RTI_APP_GEN__LIB_HelloWorldAppLibrary
```

(continues on next page)

(continued from previous page)

```
};
const struct APPGEN_LibraryModelSeq HelloWorld_libraries_sequence =
        REDA_DEFINE_SEQUENCE_INITIALIZER_W_LOAN(
                HelloWorld_libraries,
                1,
                1,
                struct APPGEN_LibraryModel);
APPGENDllExport const struct APPGEN_LibraryModelSeq*
        APPGEN_get_library_seq(void)
{
    return &HelloWorld_libraries_sequence;
}
```

4.7 Transports

4.7.1 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 domain-participants 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.

4.7.2 Transport Registration

Before a transport may be used for communication in *RTI Connext DDS Micro*, it must be registered and added to the discovery and user_traffic QoS policies. Please refer to *discovery* for details.

4.7.3 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.

4.7.4 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.

4.7.5 INTRA Transport

The built-in 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 built-in 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() callback, 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() callback 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.

4.7.6 Shared Memory Transport (SHMEM)

This section describes the optional built-in *RTI Connext DDS Micro* SHMEM transport and how to configure it.

Shared Memory Transport (SHMEM) is an optional transport that can be used in *Connext DDS Micro*. It is part of a standalone library that can be optionally linked in.

Currently, *Connext DDS Micro* supports the following functionality:

- Unicast
- Configuration of the shared memory receive queues

Registering the SHMEM Transport

The built-in SHMEM transport is a *RTI Connext DDS Micro* component that needs to be registered before a *DomainParticipant* can be created with the ability to send data across shared memory. Unlike the UDP Transport, this transport is not automatically registered. Register the transport using the code snippet below:

```
#include "netio_shmem/netio_shmem.h"

...

{
    DDS_DomainParticipantFactory *factory = NULL;
    RT_Registry_T *registry = NULL;
    struct NETIO_SHMEMInterfaceFactoryProperty shmem_property = NETIO_
↪SHMEMInterfaceFactoryProperty_INITIALIZER;
    struct DDS_DomainParticipantQos dp_qos = DDS_DomainParticipantQos_INITIALIZER;

    /* Optionally configure the transport settings */
    shmem_property.received_message_count_max = ...
    shmem_property.receive_buffer_size = ...
    shmem_property.message_size_max = ...

    registry = DDS_DomainParticipantFactory_get_registry(
            DDS_DomainParticipantFactory_get_instance());

    factory = DDS_DomainParticipantFactory_get_instance();
    registry = DDS_DomainParticipantFactory_get_registry(factory);
    if (!RT_Registry_register(
            registry,
            "_shmem",
            NETIO_SHMEMInterfaceFactory_get_interface(),
            (struct RT_ComponentFactoryProperty*)&shmem_property,
            NULL))
    {
        /* ERROR */
    }

    /* Enable the transport on a Domain Participant */
    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) = DDS_String_
↪dup("_shmem");

    DDS_StringSeq_set_maximum(&dp_qos.discovery.enabled_transports,1);
    DDS_StringSeq_set_length(&dp_qos.discovery.enabled_transports,1);
    *DDS_StringSeq_get_reference(&dp_qos.discovery.enabled_transports,0) = DDS_String_
↪dup("_shmem://");
```

(continues on next page)

(continued from previous page)

```
    DDS_StringSeq_set_maximum(&dp_qos.user_traffic.enabled_transports,1);
    DDS_StringSeq_set_length(&dp_qos.user_traffic.enabled_transports,1);
    *DDS_StringSeq_get_reference(&dp_qos.user_traffic.enabled_transports,0) = DDS_String_
↪dup("_shmem://");

    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("_
↪shmem://");

    ...

    /* Explicitly unregister the shared memory transport before clean up */
    if (!RT_Registry_unregister(
            registry,
            "_shmem",
            NULL,
            NULL)
    {
        /* ERROR */
    }
}
```

The above snippet will register a transport with the default settings. To configure it, change the invidiual configurations as described in *SHMEM Configuration*.

When a component is registered, the registration takes the properties and a listener as the 3rd and 4th parameters. The registration of the shared memory component will make a copy of the properties configurable within a shared memory transport. There is currently no support for passing in a listener as the 4th parameter.

It should be noted that the SHMEM 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.

While it is possible to register multiple SHMEM transports, it is not possible to use multiple SHMEM transports within the same participant. The reason is that SHMEM port allocation is not synchronized between transports.

Threading Model

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

- A unicast receive thread for discovery data
- A unicast receive thread for user data

Each receive thread will create a shared memory segment that will act as a message queue. Other *DomainParticipants* will send RTPS message to this message queue.

This message queue has a fixed size and can accommodate a fixed number of messages (*received_message_count_max*) each with a maximum payload size of (*message_size_max*). The total size of the queue is configurable with (*receive_buffer_size*).

Configuring SHMEM Receive Threads

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

```
struct SHMEM_InterfaceFactoryProperty shmem_property = NETIO_SHMEMInterfaceFactoryProperty_
↪INITIALIZER

shmem_property.recv_thread_property.options = ...;

/* The stack-size is platform dependent, it is passed directly to the OS */
shmem_property.recv_thread_property.stack_size = ...;

/* The priority is platform dependent, it is passed directly to the OS */
shmem_property.recv_thread_property.priority = ...;

if (!RT_Registry_register(registry, "_shmem",
                          SHMEM_InterfaceFactory_get_interface(),
                          (struct RT_ComponentFactoryProperty*)&shmem_property,
                          NULL))
{
    /* ERROR */
}
```

SHMEM Configuration

All the configuration of the SHMEM transport is done via the struct SHMEM_InterfaceFactoryProperty structure:

```
struct NETIO_SHMEMInterfaceFactoryProperty
{
    struct NETIO_InterfaceFactoryProperty _parent;
    /* Max number of received message sizes that can be residing
       inside the shared memory transport concurrent queue
     */
    RTI_INT32 received_message_count_max;
    /* The size of the receive socket buffer */
    RTI_INT32 receive_buffer_size;
    /* The maximum size of the message which can be received */
    RTI_INT32 message_size_max;
    /* Thread properties for each receive thread created by this
       NETIO interface.
     */
    struct OSAPI_ThreadProperty recv_thread_property;
};
```

received_message_count_max

The number of maximum RTPS messages that can be inside a receive thread's receive buffer. By default this is 64.

receive_buffer_size

The size of the message queue residing inside a shared memory region accessible from different processes. The default size is ((*received_message_count_max* * *message_size_max*) / 4)

message_size_max

The size of an RTPS message that can be sent across the shared memory transport. By default this number is 65536.

recv_thread_property

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

Caveats

Leftover shared memory resources

Connext DDS Micro implements the shared memory transport and utilizes shared memory semaphores that can be used conccurently by processes. *Connext DDS Micro* implements a shared memory mutex from a shared memory semaphore. If an application exits ungracefully, then the shared memory mutex may be left in a state that prevents it from being used. This can occurs because the *Connext DDS Micro* Shared Memory Transport tries to re-use and clean up and leftover segments as a result of an applications ungraceful termination. If ungraceful termination occurs, the leftover shared memory mutexes need to be cleaned up either manually or by restarting the system.

The same applies to shared memory semaphores. If an application exists ungracefully, there can be leftover shared memory segments.

Darwin-based and Linux-based systems

In the case of Darwin and Linux based systems which use SysV semaphores, you can view any leftover shared memory segments using **ipcs -a**. They can be removed using the **ipcrm** command. Shared memory keys used by *Connext DDS Micro* are in the range of 0x00400000. For example:

- `ipcs -m | grep 0x004`

The shared semaphore keys used by *Connext DDS Micro* are in the range of 0x800000; the shared memory mutex keys are in the range of 0xb00000. For example:

- `ipcs -m | grep 0x008`
- `ipcs -m | grep 0x00b`

QNX-based systems

QNX systems use Posix APIs to create shared memory segments or semaphores. The shared memory segment resources are located in **/dev/shmem** and the shared memory mutex and semaphores are located in **/dev/sem**.

To view any leftover shared memory segments when no *Connext DDS Micro* applications are running:

- `ls /dev/shmem/RTIOsapi*`
- `ls /dev/sem/RTIOsapi*`

To clean up the shared memory resources, remove the files listed.

Windows-based and VxWorks-based systems

Once all the processes that are attached to a shared memory segment, shared memory mutex, or shared memory semaphores are terminated (either gracefully or ungracefully), the shared memory resources will be automatically cleaned up by the operating system.

4.7.7 UDP Transport

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

The built-in 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 built-in UDP transport is a *RTI Connext DDS Micro* component that is automatically registered when the DDS_DomainParticipantFactory_get_instance() method is called. In order 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);

```

(continues on next page)

(continued from previous page)

```
/* The built-in 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;

/* 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 built-in 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 , and 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. *RTI 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 */
```

(continues on next page)

(continued from previous page)

```
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 */
```

(continues on next page)

(continued from previous page)

```
    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;

    /* 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 8KB. Note that *Connext DDS Micro* does not support fragmentation.

multicast_ttl

The Time-To-Live (TTL) 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)->
```

(continues on next page)

(continued from previous page)

```
                            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)->
                            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 *RTI 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 is 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 does 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, its important 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 *DomainParticipant* 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 *RTI 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;
};
```

(continues on next page)

(continued from previous page)

```
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
 * \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.
     */
```

(continues on next page)

(continued from previous page)

```
    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.
 *
 * \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;
```

(continues on next page)

(continued from previous page)

```
    /*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.
     */
    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
```

(continues on next page)

(continued from previous page)

```
     * 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 */
    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.
 *
```

(continues on next page)

(continued from previous page)

```
 * \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)
{
    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)
```

(continues on next page)

(continued from previous page)

```
{
    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
 * \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
```

(continues on next page)

(continued from previous page)

```
 *
 *
 * \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);
    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);
```

(continues on next page)

(continued from previous page)

```
    *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);
    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
```

(continues on next page)

(continued from previous page)

```
 * \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;

    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;
```

(continues on next page)

(continued from previous page)

```
}

/*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,
                       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;
```

(continues on next page)

(continued from previous page)

```
    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 =
            (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
};
```

(continues on next page)

(continued from previous page)

```
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
 */
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");
```

(continues on next page)

(continued from previous page)

```
    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;
    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)
```

(continues on next page)

(continued from previous page)

```
{
    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;
}

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
```

(continues on next page)

(continued from previous page)

```
    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))
{
    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(
```

(continues on next page)

(continued from previous page)

```
            &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)
    {
        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
     */
```

(continues on next page)

(continued from previous page)

```
    *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;
    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);
```

(continues on next page)

(continued from previous page)

```
    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;
    }

    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);
```

(continues on next page)

(continued from previous page)

```
    }

    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(
                            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 a 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 performs 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 performs 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 performs transformations and are used for discovery. Ua1 and Ub1 performs transformations and are used for user-data.

OS Configuration

In systems with serveral 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 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, command `ip route add <string> dev <interface>` can be used.

By typing command `ip route add < b0 ip >/32 dev a0` in Node A, the OS will send all packets to Node B 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.

4.8 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.

4.8.1 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 Chapter 14 Discovery section of the RTI Connext DDS Core Libraries User's Manual.

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 14.3 Discovery Implementation section of the RTI Connext DDS Core Libraries User's Manual for more details about the discovery process.

4.8.2 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.

4.8.3 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.

4.8.4 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;

/* 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*.

4.9 Generating Type Support with rtiddsgen

4.9.1 Why Use rtiddsgen?

For *RTI Connext DDS Micro* to publish and subscribe 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.

4.9.2 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 3.3 Creating User Data Types with IDL section of the RTI Connext DDS Core Libraries User's Manual for *RTI Connext DDS*.

4.9.3 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):

```
> cd $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):

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

Notes on command-line options

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

It is possible to list all command-line options specifically supported by *rtiddsgen* for *Connext DDS Micro*, by doing:

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

Existing users might notice that that previously available options, `-language microC` and `-language microC++`, have now 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.

4.9.4 Using custom data-types in an RTI Connext DDS Micro Application

An *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());
```

(continues on next page)

(continued from previous page)

```
if (retcode != DDS_RETCODE_OK)
{
    /* Log an error */
    goto done;
}

application->topic =
    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.

4.9.5 Customizing generated code

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

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

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

If none of these two options is supplied to *rtiddsgen*, they will be both considered active, and code for both *DataReaders* and *DataWriters* will be generated. If any of the two is specified then only those passed as command line are enabled.

4.9.6 Unsupported Features of rtiddsgen with RTI Connext DDS Micro

RTI 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*.

Of note, generation of example publisher/subscriber code and makefiles to compile generated files is not yet available when targeting *Connext DDS Micro*.

4.10 Threading Model

4.10.1 Introduction

This document 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*.

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

4.10.2 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
```

4.10.3 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.

4.10.4 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.

4.11 Batching

4.11.1 Introduction

This section is organized as follows:

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

4.11.2 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 RTI Connext DDS Core Libraries User's Manual.

4.11.3 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.

4.11.4 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.

4.11.5 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>
```

(continues on next page)

(continued from previous page)

```
            <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>
        <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.

4.12 Sending Large Data

4.12.1 Introduction

RTI Connext DDS Micro supports transmission and reception of data types that exceed the maximum message size of a transport. This document describes the behavior and the configuration options.

This section describes the support for:

- *Overview*
- *Configuration of Large Data*
- *Limitations*

4.12.2 Overview

RTI Connext DDS Micro supports transmission and reception of data samples that exceed the maximum message size of a transport. For example, UDP supports a maximum user payload of 65507 bytes. In order to send samples larger than 65507 bytes, *Connext DDS Micro* must split the sample into multiple UDP payloads.

When a sample larger than the transport size if sent, *Connext DDS Micro* splits the sample into fragmants and starts sending fragments based based on a flow-control algorithm. A bandwidth allocation parameters on the *DataWriter* and the scheduling rate determined how frequent and how much data can be sent each period.

When a sample is received in multiple fragments, the receiver reassmbles the each fragment into a complete serialized sample. The serialized data is then de-serialized and made available to the user as regular data.

When working with large data, it is important to keep the following in mind:

- Fragmentation is always enabled.
- Fragmentation is per *DataWriter*.
- Flow-control is per *DataWriter. It is important to keep this in mind since in |rti_core_pro| the flow-controller works accross all *DataWriters* in the same publisher.
- Fragmentation is on a per sample basis. That is, two samples of the same type may lead to fragmentation of one sample, but not the other. The application is never exposed to fragments.
- It is the *DataWriters* that determines the fragmentation size, and different *DataWriters* can use different fragmentation sizes for the same type.
- All fragments must be received before a sample can be re-constructed. When using best-effort, if a fragment is lost, the entire sample is lost. When using reliability, a fragment that is not received may be resent. If a fragment is no longer available, the entire sample is dropped.
- If one of the DDS **write()** APIs are called too fast when writing large samples, *Connext DDS Micro* may run out of resources. This is because sample may take a long time to send, and resources are not freed until the complete sample has been sent.

It is important to distinguish between the following concepts:

- Fragmentation by *RTI Connext DDS Micro*
- Fragmentation by an underlying transport, e.g. IP fragmentation when UDP datagrams exceeds about 1488 bytes.
- The maximum transmit message size of the sender. This is the maximum size of any payload going over the transport.
- The maximum transport transmit buffer size of the sender. This is the maximum number of bytes that can be stored by the transport.
- The maximum receive message size of a receiver. This is the maximum size of a single payload on a transport.
- The maximum receive buffer size of a receiver. This is the maximum number of bytes which can be received.

4.12.3 Configuration of Large Data

For a general overview of writing large data, please refer to the RTI Connext DDS Core Libraries User's Manual and the following sections:

- 6.4.1 ASYNCHRONOUS_PUBLISHER QoSPolicy
- 6.6 FlowControllers

NOTE: *Connext DDS Micro* only supports the default flow-controller.

The asynchronous publishing in is handled by a seperate that runs at fixed rate. The rate and thread properties of this thread can be adjusted in the OSAPI_SystemProperty and the following fields before DomainParticipantFactory_get_instance() is called.

```
struct OSAPI_SystemProperty sys_property = OSAPI_SystemProperty_INITIALIZER;
DDS_DomainParticipantFactory *factory = NULL;

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

sys_property.task_scheduler.thread.stack_size = ....
sys_property.task_scheduler.thread.options = ....
sys_property.task_scheduler.thread.priority = ....
sys_property.task_scheduler.rate = rate in nanosec;

if (!OSAPI_System_set_property(&sys_property))
{
    /* error */
}

factory = DDS_DomainParticipantFactory_get_instance();

....
```

4.12.4 Limitations

The following are known limitations and issues with Large Data support:

- It is not possible to disable fragmentation support.
- The scheduler thread accuracy is based on the operating system.

4.13 Zero Copy Transfer Over Shared Memory

4.13.1 Introduction

This section is organized as follows:

- *Overview*
- *Getting Started*
- *Synchronization of Zero Copy Samples*
- *Caveats*
- *Further Information*

4.13.2 Overview

Zero Copy transfer over shared memory allows large samples to be transmitted with a minimum number of copies. These samples reside in a shared memory region accessible from multiple processes. When creating a FooDataWriter that supports Zero Copy Transfer of user samples, a sample must be created with a new non-DDS API (**FooDataWriter_get_loan(...)**). This will return a pointer A* to a sample **Foo** that lies inside a shared memory segment. A reference to this sample will be sent to a receiving FooDataReader across the shared memory. This FooDataReader will attach to a shared memory segment and a pointer B* to sample **Foo** will be presented to the user. Because the two processes shared different memory spaces, A* and B* will be different but they will point to the same place in RAM.

This feature requires the usage of new RTI DDS Extension APIs:

- **FooDataWriter_get_loan()**
- **FooDataWriter_discard_loan()**
- **FooDataReader_is_data_consistent()**

For detailed information, see the C API Reference and C++ API Reference.

4.13.3 Getting Started

To enable Zero Copy transfer over shared memory, follow these steps:

1. Annotate your type with the @transfer_mode(SHMEM_REF) annotation. Currently, variable-length types (strings and sequences) are not supported for types using this transfer mode when the language binding is INBAND.

```
@transfer_mode(SHMEM_REF)
struct HelloWorld {
    long id;
    char raw_image_data[1024 * 1024]; // 1 MB
};
```

2. Register the Shared Memory Transport (see *Registering the SHMEM Transport*). References will be sent across the shared memory transport.

3. Create a FooDataWriter for the above type.

4. Get a loan on a sample using **FooDataWriter_get_loan()**.

5. Write a sample using **FooDataWriter_write()**.

For more information, see the example HelloWorld_zero_copy, or generate an example for a type annotated with @transfer_mode(SHMEM_REF):

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

Writer Side

Best practice for writing samples annotated with @transfer_mode(SHMEM_REF):

```
for (int i = 0; i < 10; i++)
{
    Foo* sample;
    DDS_ReturnCode_t dds_rc;
    /* NEW API
       IMPORTANT - call get_loan each time when writing a NEW sample
     */
    dds_rc = FooDataWriter_get_loan(hw_datawriter, &sample);

    if (dds_rc != DDS_RETCODE_OK)
    {
        printf("Failed to get a loan\n");
        return -1;
    }

    /* After this function returns with DDS_RETCODE_OK,
     * the middleware owns the sample
     */
    dds_rc = FooDataWriter_write(hw_datawriter, sample, &DDS_HANDLE_NIL);
}
```

Reader Side

```
DDS_ReturnCode_t dds_rc;
dds_rc = FooDataReader_take(...)

/* process sample here */

dds_rc = FooDataReader_is_data_consistent(hw_reader,
                                           &is_data_consistent,
                                           sample,sample_info);

if (dds_rc != DDS_RETCODE_OK)
{
    /* Error occurred when performing the consistency check */
}

if (is_data_consistent)
{
    /* Sample is consistent. Processing of sample is valid */
}
else
{
    /* Sample is NOT consistent. Any processing of the sample should
     * be discarded and considered invalid.
     */
}
```

4.13.4 Synchronization of Zero Copy Samples

There is **NO** synchronization of a zero copy sample between a sender (*DataWriter*) and receiver (*DataReader*) application. It is possible for a sample's content to be invalidated before the receiver application actually has had a chance to read it.

To illustrate this scenario, consider creating the case of creating a *Best-effort DataWriter* with max_samples of **X=1**. When the *DataWriter* is created the middleware will pre-allocate a pool of **X+1** (2) samples residing in a shared memory region. This pool will be used to loan samples when calling **FooDataWriter_get_loan(...)** ,

```
DDS_ReturnCode_t ddsrc;
Foo* sample;

ddsrc = FooDataWriter_get_loan(dw, &sample); /* returns pointer to sample 1 */
sample->value = 10000;
ddsrc = FooDataWriter_write(datawriter, sample, &DDS_HANDLE_NIL);
/*
 * Because the datawriter is using best effort, the middleware immediately
 * makes this sample available to be returned by another FooDataWriter_get_loan(...
 */

ddsrc = FooDataWriter_get_loan(dw, &sample); /* returns pointer to sample 2 */
sample->value = 20000;
ddsrc = FooDataWriter_write(datawriter, sample, &DDS_HANDLE_NIL);
/*
 * Because the datawriter is using best effort, the middleware immediately
 * makes this sample available to be returned by another FooDataWriter_get_loan(...
 */

/*
 * At this point, it is possible the sample has been received by the receiving␣
↪application
 * but has not been presented yet to the user.
 */

ddsrc = FooDataWriter_get_loan(dw, &sample); /* returns pointer to sample 1 */
/* sample->value will contain the integer 10000 because we are re-using samples
 * from a list that contains only 2 buffers.
 *
 * Also, at this point in time a referemce to sample 1 and 2 may have already been␣
↪received
 * by the middleware on the DataReader side and are lying inside a DataReader's internal␣
↪cache.
 * However, the sample may not have been received by the
 * application. If at this point the sample's value (sample->value) was changed to 999,
 * the sample returned from the Subscribers
 * read(...) or take(...) would contain unexpected values (999 instead of 10000). This␣
↪is because
 * both the Publisher and the Subscriber process have mapped into their virtual
 * address space the same shared memory region where the sample lies.
 *
 * Use **FooDataReader_is_data_consistent** to verify the consistency, to prevent this
 * scenario.
 *
 * Note, a sample is actually invalidated right after the completion
 * of FooDataWriter_get_loan(dw, &sample). If the address of the newly created sample␣
↪has been
```

(continues on next page)

(continued from previous page)

```
 * previously written and its contents has not been read by the receiver application,
 * then the previously written sample has been invalidated.
 */

ddsrc = FooDataWriter_write(datawriter, sample, &DDS_HANDLE_NIL);
```

4.13.5 Caveats

- After you call **FooDataWriter_write()**, the middleware takes ownership of the sample. It is no longer safe to make any changes to the sample that was written. If, for whatever reason, you call **FooDataWriter_get_loan()** but never write the sample, you must call **FooDataWriter_discard_loan()** to return the sample back to FooDataWriter. Otherwise, subsequent **FooDataWriter_get_loan** may fail, because the FooDataWriter has no samples to loan.
- The current maximum supported sample size is a little under the maximum value of a signed 32-bit integer. For that reason, do not use any samples greater than 2000000000 bytes.

4.13.6 Further Information

- For more information about Zero Copy transfer over shared memory see the Connext DDS Core Libraries User's Manual, Chapter 22.5 Zero Copy transfer over shared memory.

4.14 FlatData Language Binding

4.14.1 Introduction

This section is organized as follows:

- *Overview*
- *Getting Started*
- *Further Information*

4.14.2 Overview

RTI Connext DDS Micro supports the FlatData™ language binding in the same manner as *RTI Connext DDS*. However, *RTI Connext DDS Micro* only supports FlatData language binding for traditional C++ APIs, whereas *RTI Connext DDS* also supports it for the Modern C++ API. FlatData language binding is not supported for the C language binding.

4.14.3 Getting Started

The best way to start is to generate an example by creating an example IDL file HelloWorld.idl containing the following IDL type:

```
@final
@language_binding(FLAT_DATA)
```

(continues on next page)

(continued from previous page)

```
struct HelloWorld
{
   long a;
}
```

Next, run:

`rtiddsgen -example -micro -language C++ HelloWorld.idl`

4.14.4 Further Information

For more details about FlatData language binding as a feature, please refer to the Connext DDS Core Libraries User's Manual Chapter 22.4 FlatData language binding.

For documentation about how to build and read a FlatData sample, see Pro_Flat_Data_API.

4.15 Security SDK

4.15.1 Introduction

RTI Security Plugins introduce a robust set of security capabilities, including authentication, encryption, access control and logging. Secure multicast support enables efficient and scalable distribution of data to many subscribers. Performance is also optimized by fine-grained control over the level of security applied to each data flow, such as whether encryption or just data integrity is required.

The *RTI Connext DDS Micro Security SDK* includes a set of builtin plugins that implement the Service Plugin Interface defined by the DDS Security specification.

Security Plugins are available in a separate package from the RTI Support Portal, https://support.rti.com/.

It is also possible to implement new custom plugins by using the Security Plugins SDK bundle (for more information, please contact s upport@rti.com). See the RTI Security Plugins Release Notes and RTI Security Plugins Getting Started Guide.

4.15.2 Installation

Please refer to the *Installation* chapter for how to install the *RTI Connext DDS Micro Security SDK*.

4.15.3 Examples

For descriptions and examples of the security configuration in this release, please consult the HelloWorld_dpde_secure examples under the example/[unix, windows]/[C, CPP] directory. *RTI Connext DDS Micro Security SDK* supports both the C and C++ programming languages.

To use the *RTI Connext DDS Micro Security SDK*, you will need to create private keys, identity certificates, governance, and permission files, as well as signed versions for use in secure authenticated, authorized, and/or encrypted communications.

4.15.4 Enabling RTI Security Plugins

In order to enable the RTI Security Plugins, the name of a "plugin suite" (i.e. the collection of security plugins defined by DDS Security) must be specified in a DomainParticipant's QoS. Plugin factories for this suite must also be registered with the RT_Registry before the DomainParticipant is created.

When using *RTI Connext DDS Micro*'s C API, this can be achieved with the following code:

```
RTI_BOOL result = RTI_FALSE;
struct DDS_DomainParticipantQos dp_qos = DDS_DomainParticipantQos_INITIALIZER;
struct SECCORE_SecurePluginFactoryProperty sec_plugin_prop = SECCORE_
↪SecurePluginFactoryProperty_INITIALIZER;
DDS_DomainParticipantFactory *factory =  DDS_DomainParticipantFactory_get_instance();
RT_Registry_T *registry =  DDS_DomainParticipantFactory_get_registry(factory);

/* Register factories for built-in security plugins, using default
 * properties and default name */
if (!SECCORE_SecurePluginFactory_register(
        registry,SECCORE_DEFAULT_SUITE_NAME,&sec_plugin_prop))
{
    printf("failed to register security plugins\n");
    goto done;
}

/* In order to enable security, the name used to register the suite of
 * plugins must be set in DomainParticipantQos */
if (!RT_ComponentFactoryId_set_name(
        &dp_qos->trust.suite, SECCORE_DEFAULT_SUITE_NAME))
{
    printf("failed to set component id\n");
    goto done;
}

result = RTI_TRUE;

done:

return result;
```

For users of *RTI Connext DDS Micro*'s C++ API, the suite can be registered using the following code:

```
RTI_BOOL result = RTI_FALSE;
DDS_DomainParticipantQos dp_qos;
SECCORE_SecurePluginFactoryProperty sec_plugin_prop;
DDSDomainParticipantFactory *factory = DDSDomainParticipantFactory::get_instance();
RTRegistry_T *registry = factory->get_registry();

/* Register factories for built-in security plugins, using default
 * properties and default name */
if (!SECCORE_SecurePluginFactory::register_suite(
        registry,SECCORE_DEFAULT_SUITE_NAME,sec_plugin_prop))
```

(continues on next page)

(continued from previous page)

```
{
    printf("failed to register security plugins\n");
    goto done;
}

/* In order to enable security, the name used to register the suite of
 * plugins must be set in DomainParticipantQos */
if (!dp_qos.trust.suite.set_name(SECCORE_DEFAULT_SUITE_NAME))
{
    printf("failed to set component id\n");
    goto done;
}

result = RTI_TRUE;

done:

return result;
```

Additional properties can be controlled using (key,value) pairs in a DomainParticipant's DDS_PropertyQosPolicy.

Configuration keys (and corresponding valid values) supported by each security plugin are listed by each plugin's section of this manual.

In *RTI Connext DDS Micro*, you must set the security-related participant properties before you create a participant. You cannot create a participant without security and then call DomainParticipant::set_qos() with security properties, even if the participant has not yet been enabled.

4.16 Building Against FACE Conformance Libraries

4.16.1 Introduction to RTI Connext DDS Micro Support for FACE

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

4.16.2 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 files. 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 www.cmake.org.

4.16.3 FACE Golden Libraries

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

Building the FACE Golden Libraries

The FACE conformance tools ships with its 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 with GCC 4.4.5. For completeness, the complete list of all the files modified by RTI are included below in source form.

4.16.4 Building the Connext DDS Micro Source

The following is step by step instructions on how to built the *RTI 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 up 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 RTI Connext DDS Micro 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.
  ```

(continues on next page)

(continued from previous page)

```
   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 librarires build 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
   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>*.

4.17 Working With Sequences

4.17.1 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.

4.17.2 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 |
     +---+
  2  | T |
     +---+
       |
       |
     +---+
N-1  | T |
     +---+
```

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

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

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

IDL Type	*RTI Connext DDS Micro* Type	*RTI 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 which 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)
```

(continues on next page)

(continued from previous page)

```
{
    *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);
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 this examples 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:

Conside 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;
```

(continues on next page)

(continued from previous page)

```
/* 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 between 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 */

```

(continues on next page)

(continued from previous page)

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

DDS_StringSeq_finalize(&my_seq);
```

4.18 Debugging

4.18.1 Overview

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

4.18.2 Configuring Logging

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

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

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

The default buffer size is different for Debug and Release libraries, with Debug libraries 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, e.g. 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 macro OSAPI_LOG_WRITE_BUFFER. This macro shall have the same parameters as function prototype OSAPI_Log_write_buffer_T.

It is also possible to set this function during runtime by using 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.

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

4.18.4 Interpreting Log Messages and Error Codes

A log entry in *RTI 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 *RTI Connext DDS Micro*.

RTI 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.
- *T* The log message has a valid timestamp (successful call to OSAPI_System_get_time()).

- *E* Exception, the log message has been truncated.

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=6" corresponds to RTPS.
- 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).".

Chapter 5

Building and Porting

5.1 Building the RTI Connext DDS Micro Source

5.1.1 Introduction

RTI Connext DDS Micro has been engineered for reasonable portability to common platforms and environments, such as Darwin, iOS, Linux, and Windows. This user manual 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 *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.

5.1.2 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 16bit 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.

5.1.3 Overview of RTI 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 a a directory called src/ under *RTI 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 manual, `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.

For the remainder of this document `RTIMEROOT` refers to both source/unix and source/windows. Only when necessary will it be pointed out whether it is the windows or Unix source that it is being referred too.

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. To link against built libraries instead of those shipped by RTI, set `RTIMEHOME` to `RTIMEROOT` (2).

NOTE 1: This applies to builds using CMake. To build in a custom environment, please refer to *Custom Build Environments*.

NOTE 2: The path to an installation of *rtiddsgen*, likely from a bundle shipped by RTI, will also have to be specified separately.

CMakeLists.txt.txt is the main input file to CMake and is used to generate build files.

The *RTIMEROOT/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 *RTIMEROOT/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 *RTIMEROOT/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 *RTIMEROOT/lib* directory is empty by default. All libraries successfully built with the CMake generated build-files, independent of which generator was used, will be copied to the *RTIMEROOT/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_netioshmem* - Shared Memory Transport
- *rti_me_netiosdm* - Zero copy over shared memory transport library
- *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* prefix 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 liked release library

5.1.4 Compiling RTI 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 *RTI 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 RTI 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 is a convenient way to invoke CMake with the correct options.

Unix:

```
RTIMEROOT/resource/script/rtime-make --type Debug --target self \
                      --name i86Linux2.6gcc4.4.5 -G "Unix Makefiles" --build
```

Windows:

```
C:RTIMEROOT\resource\scripts|rti_me|-make --config Debug --target self \
                   --name i86Win32VS2010 -G "Visual Studio 10 2013" --build
```

Explanation of arguments:

- `-config Debug` : Create Debug build.
- `-target \<target\>` : The target for the sources will be built. self indicates that the host machine is the target and *RTI Connext DDS Micro* will be built with the options that CMake automatically determines for the local compiler. Please refer to ref source_xbuild for information for more information on specifying the target architecture to build for.
- `-name \<name\>` : This is the name of the build and shall be a descriptive name following the recommendation on naming described in section ref source_prepare. If - -name is not specified, the value for - -target will be used as name.
- `-build Build` the generated project files.

On Unix:

- If gcc is part of the name GCC is assumed.
- If clang is part of the name clang is assumed.

On Windows:

- If Win32 if part of the name, a 32 bit Windows build is assumed.
- If Win64 if part of the name, a 64 bit Windows build is assumed.

To get a list of all the options:

```
rtime-make -h
```

To get a help for a specific target, use:

```
rtime-make --target <target> --help
```

Manually Building with CMake

Preparation 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 to assign 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 RTI 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 running FreeRTOS compiled with gcc 4.6.1, compiled by acme.
- acme_i86Win32VS2015 - *Connext DDS Micro* for a i386 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 a i386 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 Buildfiles for RTI 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*.

It is recommended that the GUI is started from the `RTIMEROOT` directory. If this is not the case, check that:

- The source directory is the location of `RTIMEROOT`.
- 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 RTIMEROOT/lib/<name>

The generated build-files may contain different sub-projects that are specific to the tool. For example, in Xcode and MS Visual Studio the following targets are available:

- ALL_BUILD - Builds all the projects.
- rti_me_<name> - Build 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 where generated:

```
make help
```

Creating Buildfiles for RTI Connext DDS Micro Using CMake from The Command-line

Open a terminal window in the `RTIMEROOT` 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 *RTIMEROOT/lib/<name>*.

The generated build-files may contain different sub-projects that are specific to the tool. For example, in Xcode and MS Visual Studio the following targets are available:

- ALL_BUILD - Builds all the projects.
- rti_me_<name> - Build 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 where generated:

```
make help
```

CMake Flags used by RTI Connext DDS Micro

The following CMake flags (-D) is 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 invoke cmake directly on the CMakeLists.txt provided by *Connext DDS Micro*.

- `-DRTIME_TARGET_NAME=\<name\>` - The name of the target (equivalant of `--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. Default is undefined which means shared libraries is built. NOTE: This flags 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 libraris 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 < 2.8.11 because the TIMESTAMP function does not exist.
- `-DOPENSSLHOME=<path>` to OpenSLL 1.0.1
- `-DRTIME_TRUST_INCLUDE_BUILTIN=false` Exclude the builtin security plugin from the build.
- `-DRTIME_DDS_DISABLE_PARTICIPANT_MESSAGE_DATA=false` Disables P2P Message Data inter-participant channel. This channel is needed to use **DDS_AUTOMATIC_LIVELINESS_QOS** and **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS** with finite lease duration.

5.1.5 RTI Connext DDS Micro Compile Options

The *Connext DDS Micro* source supports compile-time options. The compile time 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 not longer possible to build a single library using CMake. Please refer to *Custom Build Environments* for information on customized builds.

RTI 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 - including logging based on the default rule.
- **OSAPI_ENABLE_TRACE** (Include/Exclude/Default)
  - Include - Include tracing.
  - Exclude - Exclude tracing.
  - Default - including tracing based on the default rule.
- **OSAPI_ENABLE_PRECONDITION** (Include/Exclude/Default)
  - Include - Include tracing.
  - Exclude - Exclude tracing.
  - Default - including precondition checks based on the default rule.

RTI Connext DDS Micro Platform Selection

The *Connext DDS Micro* build system looks for target platform files in *RTIMEROOT/include/osapi*. All files that matches *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.

RTI Connext DDS Micro Compiler Selection

The *Connext DDS Micro* build system looks for target compiler files in *RTIMEROOT/include/osapi*. All files that matches *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 compiler this this may not work. Instead the appropriate compiler definition file can be selected here.

RTI 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 UDP transport.

Checking the **RTIME_UDP_EXCLUDE_BUILTIN** removes UDP transport from being built.

5.1.6 Cross Compiling RTI Connext DDS Micro

Cross-compiling the *Connext DDS Micro* source-code uses the exact same process described in *Compiling RTI 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 *RTI 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. The VxWorks tool-chain file uses the WindRiver 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 the *RTIMEROOT/source/Unix/resource/CMake/architectures* or *RTIMEROOT/source/windows/resource/CMake/architectures* directory.

Once as 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
```

5.1.7 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 is 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 RTI 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 lets you choose if 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 defintion `-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"`

5.2 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 are 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*.

5.2.1 Updating from RTI 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 preprocesor 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 needs to be ported.

5.2.2 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 RTI Connext DDS Micro Source* for how to build the source code.

This 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
                                      |
                                      +-- <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 files properly detects GCC and MSVC and it is not necessary to provide a new file if one of these compilers are 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.

5.2.3 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.

5.2.4 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*.

5.2.5 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*

5.2.6 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*.

5.2.7 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*.

5.2.8 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.

5.2.9 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

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 7

Copyrights

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 8

Release Notes

8.1 Supported Platforms and Programming Languages

Connext DDS Micro supports the C and traditional C++ language bindings.

Connext DDS Micro is available as pre-built binaries for the following platforms:

- *Supported Linux Platforms*
- *Supported Windows Platforms*
- *Supported VxWorks Platforms*

Note that RTI only tests on a subset of the possible combinations of OS and CPU. Please refer to the following tables for a list of specific platforms and the specific configurations that are tested by RTI.

Table 8.1: Supported Linux Platforms

OS	CPU	Compiler	RTI Architecture Abbreviation
Red Hat Enterprise Linux 6.0, 6.1 (2.6 kernel)	x86	gcc 4.4.5	i86Linux2.6gcc4.4.5

Table 8.2: Supported Windows Platforms

OS	CPU	Compiler	RTI Architecture Abbreviation
Windows 7	x86	Visual Studio 2010	i86Win32VS2010

Table 8.3: Supported VxWorks Platforms

OS	CPU	Compiler	RTI Architecture Abbreviation
VxWorks 6.9	Pentium 32-bit	gcc 4.3.3	For Kernel Modules: pentiumVx6.9gcc4.3.3 For Real Time Processes: pentiumVx6.9gcc4.3.3_rtp
VxWorks 6.9	PPC32	gcc 4.3.3	For Kernel Modules: ppc604Vx6.9gcc4.3.3 For Real Time Processes: ppc604Vx6.9gcc4.3.3_rtp
VxWorks Cert 6.6.4.1	PPC32 e500v2	gcc 4.1.2	ppce500v2VxCert6.6.4.1gcc4.1.2

8.2 Compatibility

For backward compatibility information between 3.0.0 and previous releases, see the *Migration Guide* on the RTI Community Portal (https://community.rti.com/documentation).

8.3 What's New in 3.0.0

8.3.1 Support for XCDR encoding version 2

This release adds support for the standard XCDR encoding version 2 data representation described in the "Extensible and Dynamic Topic Types for DDS" specification. This encoding version is more efficient in terms of bandwidth than XCDR encoding version 1, which is supported in previous *Connext DDS* releases (and still supported in this release).

To select between XCDR and XCDR2 data representations, you can use the DataRepresentationQosPolicy for DataReaders and DataWriters. *Connext DDS Micro* now supports this policy. You may specify XCDR, XCDR2, or AUTO to indicate which versions of the Extended Common Data Representation (CDR) are offered and requested. The default is AUTO.

A DataWriter offers a single representation, which indicates the CDR version the DataWriter uses to serialize its data. A DataReader requests one or more representations, which indicate the CDR versions the DataReader accepts. If a DataWriter's offered representation is contained within a reader's sequence of requested representations, then the offer satisfies the request, and the policies are compatible. Otherwise, they are incompatible. In support of this feature, a new QoS, DATA_REPRESENTATION, has been added for the DataWriter and DataReader. There is also a new annotation, @allowed_data_representation, that can be used to select the supported data representations for a type.

For more information, see:

- The "Extensible and Dynamic Topic Types for DDS" specification from the Object Management Group (OMG): https://www.omg.org/spec/DDS-XTypes/.
- The 6.5.3 DATA_REPRESENTATION QoS Policy section in the RTI Connext DDS Core Libraries User's Manual.

- The Chapter 4 Data Representation section in the RTI Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types.

8.3.2 Large data streaming using RTI FlatData™ language binding and Zero Copy transfer over shared memory

To meet strict latency requirements, you can reduce the default number of copies made by the middleware when publishing and receiving large samples (on the order of MBs) by using two new features: FlatData language binding and Zero Copy transfer over shared memory.

These features can be used standalone or in combination.

Using the FlatData language binding, you can reduce the number of copies from the default of four copies to two copies, for both UDP and shared memory communications. FlatData is a language binding in which the in-memory representation of a sample matches the wire representation, reducing the cost of serialization/deserialization to zero. You can directly access the serialized data without deserializing it first. To select FlatData as the language binding of a type, annotate it with the new @language_binding(FLAT_DATA) annotation.

Zero Copy transfer over shared memory allows you to reduce the number of copies to zero for communications within the same host. This feature accomplishes zero copies by using the shared memory builtin transport to send references to samples within a shared memory segment owned by the DataWriter, instead of using the shared memory builtin transport to send the serialized sample content by making a copy. With Zero Copy transfer over shared memory, there is no need for the DataWriter to serialize a sample, and there is no need for the DataReader to deserialize an incoming sample since the sample is accessed directly on the shared memory segment created by the DataWriter. The new TransferModeQosPolicy specifies the properties of a Zero Copy DataWriter.

For more information on setting up and using one or both of these features, see the Chapter 22 Sending Large Data chapter in the RTI Connext DDS Core Libraries User's Manual.

8.3.3 Support for RTI Security Plugins

RTI Security Plugins introduce a robust set of security capabilities, including authentication, encryption, access control and logging. Secure multicast support enables efficient and scalable distribution of data to many subscribers. Performance is also optimized by fine-grained control over the level of security applied to each data flow, such as whether encryption or just data integrity is required.

Security Plugins are available in a separate package from the RTI Support Portal, https://support.rti.com/.

See the RTI Security Plugins Release Notes and RTI Security Plugins Getting Started Guide.

8.3.4 Large Data Types

This release adds support for user-defined data types that exceed the maximum message size supported by the underlying transports, such as 64K in the case of UDP. Its use is fully transparent: samples are automatically fragmented by the DataWriter and reassembled by the DataReader. Once re-assembled, the samples are treated as regular samples and subject to all applicable QoS policies.

8.3.5 Asynchronous DataWriters

This release adds support for publishing data asynchronously. An asynchronous DataWriter offloads the user thread and makes it possible to coalesce samples across multiple write() calls into a single network packet.

Samples written by an asynchronous DataWriter are not sent in the context of the user thread as part of the write() call. Instead, samples are queued and sent in the context of a separate, dedicated thread. An optional flow control mechanism is provided to throttle the rate at which samples are coalesced and sent by the dedicated thread.

To implement this feature, there are two new QosPolicies, ASYNCHRONOUS_PUBLISHER and PUBLISH_MODE. The ASYNCHRONOUS_PUBLISHER QosPolicy enables/disables asynchronous publishing for the Publisher. If enabled, the Publisher will spawn a separate asynchronous publishing thread, which will be shared by all of the Publisher's DataWriters that have their new PUBLISH_MODE QosPolicy set to ASYNCHRONOUS. When data is written asynchronously, a new 'FlowController' object can be used to shape the network traffic. The FlowController's properties determine when the asynchronous publishing thread is allowed to send data and how much.

8.3.6 Support for KEEP_ALL History

This release supports setting the History QoS policy **kind** to KEEP_ALL.

8.3.7 Support for AUTOMATIC and MANUAL_BY_PARTICIPANT Liveliness

Now you can set the Liveliness QoS policy **kind** to AUTOMATIC or MANUAL_BY_PARTICIPANT.

- AUTOMATIC: *Connext DDS Micro* will automatically assert liveliness for the DataWriter at least as often as the lease_duration.
- MANUAL_BY_PARTICIPANT: The DataWriter is assumed to be alive if any Entity within the same DomainParticipant has asserted its liveliness.

8.3.8 Micro Application Generation

This release includes Micro Application Generation, which enables you to create a *Connext DDS Micro* application, including registration of factories and creation of DDS entities, from an XML configuration file. Please refer to the Chapter 6 Generating Applications for Connext DDS Micro in the RTI Connext DDS Core Libraries XML-Based Application Creation Getting Started Guide and the Application Generation section of the RTI Connext DDS Micro User's Manual for details. The Micro Application Generation is enabled by default in this release when compiling with rtime-make. However, future releases may disable the feature by default. Thus, it is advised to always compile with the Micro Application Generation feature enabled (-DRTIME_DDS_ENABLE_APPGEN=1 to cmake).

8.3.9 Ability to use only one UDP port per DomainParticipant

This release provides a way to use just one UDP port per DomainParticipant. The advantage of this is that by only using one UDP port, *Connext DDS Micro* will only create a receive thread, so fewer resources are used, mainly stack memory.

The disadvantage is that the port mappings used are not compliant with the OMG's DDS Interoperability Wire Protocol and communication with other DDS implementations might not be possible.

You can only use this feature if multicast OR unicast is used for both discovery and user traffic. If both unicast AND multicast are configured, you cannot use this feature.

To enable this feature, assign the same value to both the builtin and user port offsets in RtpsWellKnownPorts_t.

8.3.10 New C++ DPSE example

This release includes a new C++ example that uses DPSE (dynamic participant - static endpoint) discovery.

8.4 What's Fixed in 3.0.0

8.4.1 Linker error when using shared libraries on VxWorks systems

There was a linker error when compiling examples for architecture ppc604Vx6.9gcc4.3.3 using shared libraries. The compiler reported that the libraries could not be found. This issue has been fixed.

[RTI Issue ID MICRO-1841]

8.4.2 Failure to link VxWorks RTP mode using shared libraries compiled with CMake

Wrong compiler options were used when compiling for VxWorks RTP mode using CMake. This problem caused a linker error when trying to link an application using shared libraries. This issue has been fixed.

[RTI Issue ID MICRO-1909]

8.4.3 CPU endianness detection method improved

The CPU endianness detection method has been improved. Now the CMake endian test is used. If CMake is not used to compile, the compiler preprocessor macros are used to infer CPU endianness.

[RTI Issue ID MICRO-1919]

8.4.4 Examples used untyped register_type APIs instead of typed APIs

The provided examples have been updated to use FooTypeSupport_register_type() instead of DDS_DomainParticipant_register_type(). Using the typed API to register types is preferred over using the untyped API.

[RTI Issue ID MICRO-1922]

8.4.5 Wait_set 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]

8.4.6 WaitSet waited less than specified time period

A WaitSet may have waited less than the specified time period. This problem has been resolved.

[RTI Issue ID MICRO-1950]

8.4.7 Samples with deserialization errors were accepted

In previous versions, samples that could not be deserialized was rejected, causing samples to be resent when reliability was enabled. This behavior has been changed; now samples with deserialization errors are accepted and discarded.

[RTI Issue ID MICRO-1954]

8.4.8 Potential wrong API used when using host name as peer

The getaddrinfo() API was incorrectly used when a host name was used as a peer. That error might have caused a run-time error. This problem occurred only if compilation was done for Windows or if FACE compliance was enabled. This issue has been fixed.

[RTI Issue ID MICRO-1957]

8.5 Known Issues

8.5.1 Flow Controllers require RTOS

Flow controllers require RTOS. This will be addressed in the next release.

Chapter 9

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 10

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.