# RTI Connext Cert

User's Manual

Version 2.4.15

# Contents

*RTI® Connext® Cert* 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 Cert* is designed as a certifiable component in high-assurance systems.

Key benefits of *Connext Cert* include:

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

# Chapter 1

# Introduction

## 1.1 What is RTI Connext Cert?

*RTI Connext Cert* 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 Cert* uses the publish-subscribe communications model to make data distribution efficient and robust. *Connext Cert* simplifies application development, deployment and maintenance and provides fast, predictable distribution of time-critical data over a variety of transport networks. With *Connext Cert*, 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 Cert* 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 Cert*, 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 Cert* is highly configurable so programmers can adapt it to meet the application's specific communication requirements.

### 1.1.1 Publish-Subscribe Middleware

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

## 1.2 Supported DDS Features

*Connext Cert* 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.

### 1.2.1 DDS Entity Support

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

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

### 1.2.2 DDS QoS Policy Support

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

- DDS_DataReaderProtocolQosPolicy
- DDS_DataReaderResourceLimitsQosPolicy
- DDS_DataWriterProtocolQosPolicy
- DDS_DataWriterResourceLimitsQosPolicy
- DDS_DeadlineQosPolicy

- DDS_DiscoveryQosPolicy
- DDS_DomainParticipantResourceLimitsQosPolicy
- DDS_DurabilityQosPolicy
- DDS_DestinationOrderQosPolicy
- DDS_EntityFactoryQosPolicy
- DDS_HistoryQosPolicy
- DDS_LivelinessQosPolicy
- DDS_OwnershipQosPolicy
- DDS_OwnershipStrengthQosPolicy
- DDS_ReliabilityQosPolicy
- DDS_ResourceLimitsQosPolicy
- DDS_RtpsReliableWriterProtocol_t
- DDS_SystemResourceLimitsQosPolicy
- DDS_TransportQosPolicy
- DDS_UserTrafficQosPolicy
- DDS_WireProtocolQosPolicy

## 1.3 Standards and Interoperability

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

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

*Connext Cert*, RTI *Connext Micro*, and *Connext* are wire-interoperable, unless stated otherwise (see below), and API compatible for APIs specified by the DDS standard. For non-standard APIs, *Connext Cert*, RTI *Connext Micro*, and *Connext* are incompatible. Please refer to *Working with RTI Connext Cert and RTI Connext* for more information.

### 1.3.1 DDS Wire Compatibility

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

| Submessage | Supported | DDS Standard | Connext DDS Core |
|---|---|---|---|
| DATA_FRAG | No | Yes | Yes |
| NACK_FRAG | No | Yes | Yes |
| HEARTBEAT_FRAG | No | Yes | No |
| DATA_BATCH | No | No | Yes |
| HEARTBEAT_BATCH | No | No | No |
| INFO_SRC | No | Yes | Yes |
| INFO_REPLY | No | Yes | Yes |
| INFO_REPLY_IPV4 | No | Yes | Yes |

### 1.3.2 Profile / Feature

*Connext Cert* does not support mutable Qos policies.

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

continues on next page

Table 1.1 – continued from previous page

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

NOTES:

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

### 1.3.3 DDS API Support

For supported APIs, please refer to:

- C API Reference

## 1.4 RTI Connext DDS Documentation

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

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

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

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

## 1.5 OMG DDS Specification

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

## 1.6 Other Products

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

*RTI Connext Cert* is a subset of RTI *Connext Micro*. *Connext 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()).
- Batching.
- UDP Transformations.

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

*RTI Connext Cert* is provided in two RTI target package files (`.rtipkg`):

- `rti_connext_dds_cert-<version>-host.rtipkg`
- `rti_connext_dds_cert-<version>-target-<architecture>.rtipkg`

You must first install *RTI Connext Professional* 7.3.0 before installing *Connext Cert* packages. To install the *Connext Cert* packages:

1. Open the *RTI Launcher* for *Connext Professional* 7.3.0.
2. Navigate to the **Configuration** tab.
3. Select **Install RTI Packages**.
4. In the popup window, click on the + icon and add both `.rtipkg` files to the installation queue.
5. Select **Install**.

Once installed, you will find a directory called `rti_connext_dds_cert-<version>` in the *Connext Professional* installation directory.

---

**Note:** A Java Runtime Environment (JRE) is needed to run the IDL compiler *rtiddsgen*. By default, *Connext Cert* will use the JRE that is already included in the *Connext Drive* or *Connext Professional* installation where *Connext Cert* is installed.

If you prefer a different JRE, you must set the environment variable `JREHOME` to the path of the specified JRE.

---

**Warning:** RTI strongly recommends that you copy the *Connext Cert* installation directory outside of the *Connext Professional* installation. It may not be desirable to build *Connext Cert* libraries inside the *Connext Professional* directory due to patches, lack of write access, or other factors.

## 2.2 Overview of the Host Bundle

This section provides an overview of the host package contents, as well as the resultant directory structure in the *Connext Cert* installation.

When installed, the host bundle (`rti_connext_dds_cert-<version>-host.rtipkg`) adds the following to the *Connext Cert* directory:

```
--rti_connext_dds-<version>/
  |
  +--rti_connext_dds_cert-<version>/
      |--doc/
      |--include/
      |--resource/
      |--rtiddsgen/
      |--CMakeLists.txt
      |--ReadMe.html
      +--src/
         +--rti_me_psl.1.0
```

- The `doc/` directory contains this documentation, as well as the C and C++ API References and the Safety Integration Manual.
- The `include/` directory contains the public header files to compile applications.
- The `resource/` directory contains the build system used for the examples and Platform Support Libraries (PSL).
- The `rtiddsgen/` directory contains an IDL compiler for type support code.
- CMakeLists.txt is the main input file to CMake and is used to generate build files.
- `ReadMe.html` opens this documentation.
- `src/` contains the source files for all supported Platform Support Libraries (PSL); refer to *Building the PSL* and the Safety Integration Manual for more information on building the PSL for your platform.

## 2.3 Overview of the Target Bundle

This section provides an overview of the target package contents, the types of libraries included in the bundle, and the library names and descriptions.

When installed, the target bundle (`rti_connext_dds_cert-<version>-target-<architecture>.rtipkg`) adds the following to the *Connext Cert* directory:

```
--rti_connext_dds-<version>/
  |
  +--rti_connext_dds_cert-<version>/
      |
      +--lib/
          +--<arch>CERT
          |   +---<arch libraries>
```

- The `lib/` directory contains the libraries needed to build *Connext Cert* applications.
  - `<arch>CERT` contains pre-built static CERT profile Release and Debug libraries.

### 2.3.1 Library types

*Connext Cert* provides precompiled binaries for supported architectures. This section explains the different library types and gives a general description of the binaries shipped by RTI.

In this section, the following terms are used:

- *toolchain* refers to the compiler, linker, and archiver for a specific CPU architecture, excluding dependencies in standard header files and libraries.
- *platform* refers to the hardware, BSPs, OS kernel, and C/C++ libraries that are not included in the toolchain (such as `libc`, `libc++`, and the network stack).

*Connext Cert* consists of two libraries: a Platform Independent Library (PIL) and a Platform Support Library (PSL). The PIL is an `rti_me` library that includes all functionality for *Connext Cert* except for platform integration code. The PSL is a library that supports OS integration and network stack integration, including transports (such as UDPv4) and mutex and semaphore support.

This is illustrated below:

The main benefit of splitting *Connext Cert* functionality into two libraries is that the PIL does not need to be recertified for every platform. PSLs can be written for the same PIL without having to recompile or recertify the platform-independent code.

RTI provides the PIL as a pre-compiled binary in the target bundle, as described above. However, RTI does not provide pre-compiled PSLs for *Connext Cert*; you must build the PSL for your target architecture from the source code provided in the *Host Bundle*. Refer to *Building the PSL* and the Safety Integration Manual for instructions on how to build the PSL.

**Note:** The PIL is compiled without standard C header files and is only dependent on the toolchain. This is different from the integrated libraries, which are compiled with standard C header files.

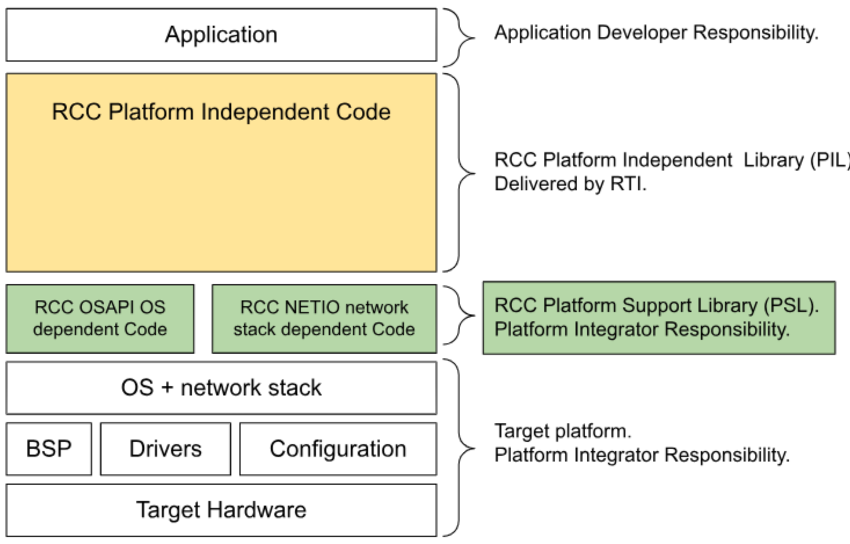

Application
RCC Platform Independent Code
RCC OSAPI OS dependent Code
RCC NETIO network stack dependent Code
OS + network stack
BSP
Drivers
Configuration
Target Hardware
Application Developer Responsibility.
RCC Platform Independent Library (PIL). Delivered by RTI.
RCC Platform Support Library (PSL). Platform Integrator Responsibility.
Target platform. Platform Integrator Responsibility.

Figure 2.1: An overview of *Connext Cert* (RCC) libraries

The PSL must be compiled against the standard C header files, as well as other platform-dependent header files.

### 2.3.2 Library descriptions

The following libraries are included in the target bundle. Note that the names listed below do not include platform-specific prefixes or suffixes.

Depending on the target architecture, the library name is prefixed with lib and the library suffix also varies between target architectures.

The following naming conventions are also used:

- Static libraries have a z suffix.
- Debug libraries have a d suffix.
- Release libraries do not have an additional suffix.

For example, `rti_mezd` indicates a static debug library, while `rti_mez` indicates a static release library.

Table 2.1: Target Bundle Libraries

| Library Name | Description |
|---|---|
| `rti_me` | Includes the following:<br>• The core functionality for *Connext Cert*, including the DDS C API.<br>• The Dynamic Participant Static Endpoint (DPSE) plugin.<br>• The Reader and Writer History plugins. |
| `rti_me_netiozcopy` | Includes the Zero Copy v2 transport. |

## 2.4 Directory Structure

The complete, default *Connext Cert* installation is structured as shown below:

```
+--rti_connext_dds-<version>/
   |
   +--rti_connext_dds_micro-<version> (rti_connext_dds_micro-<version>-host.rtipkg)
      |--doc/
      |--include/
      |--resource/
      |--rtiddsgen/
      |--CMakeLists.txt
      |--ReadMe.html
      +--src/
      |  +-- rti_me_psl.1.0
      +--lib (rti_connext_dds_micro-<version>-target-<arch>.rtipkg)
         +--<arch>CERT
         |   +---<arch libraries>
```

# Chapter 3

# Building Connext Cert

Before you get started with *RTI Connext Cert*, you must build the Platform Support Library (PSL) for your platform from the source code provided in your *Connext Cert* installation. This section describes how to compile the PSL for an architecture supported by RTI (see *Supported Platforms and Programming Languages* for more information).

---

**Note:** *Connext Cert* also requires a Platform Independent Library (PIL), which is provided as a pre-compiled binary in your *Connext Cert* installation and does not need to be built.

---

This section is written for developers and engineers with a background in software development. RTI recommends reading this section in order, as one subsection may refer to or assume knowledge about concepts described in a preceding subsection.

## 3.1 Setting up the build environment

The following terminology is used to refer to the environment in which *Connext Cert* is built and run:

- The *host* is the machine that runs the software to compile and link *Connext Cert*.
- The *target* is the machine that runs *Connext Cert*.
- In many cases *Connext Cert* 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 Cert*.

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 Cert* is built, used, or run.

The word *may* is used to describe an optional feature.

### 3.1.1 The host environment

*Connext Cert* 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 Cert* 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 Cert*, and in some cases it may also not be recommended. As a rule of thumb, if *Connext Cert* 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).

Supported host environments are Windows (cygwin and mingw are not tested), Linux, and macOS systems.

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.

### 3.1.2 The target environment

The target machine must:

- support 8-bit, 16-bit, and 32-bit signed and unsigned integers. Note that a 16-bit CPU (or even 8-bit) is supported as long as the listed types are supported.

  *Connext Cert* 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.

The remainder of this manual assumes that the target environment is one supported by RTI:

- POSIX (Linux, macOS, QNX®, VOS, iOS, Android)
- VxWorks 6.9 or later
- Windows
- QNX

## 3.2 Building the PSL

There are two recommended methods to compile the PSL: by running the `rtime-make` script (which invokes CMake), or by invoking CMake manually. Both are described in more detail below.

CMake is the preferred tool to build *Connext Cert* because it simplifies configuring the *Connext Cert* 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 Cert*. This system makes it easier to support building *Connext Cert* in different build environments. CMake is easy to install with pre-built binaries for common environments and has no dependencies on external tools.

Alternatively, you can include the PSL source as part of a *Connext Cert* application. Refer to *Custom build environments* for more information on this option.

### 3.2.1 Building the PSL with rtime-make

The *Connext Cert* source bundle includes a bash (UNIX) and BAT (Windows) script to simplify the invocation of CMake. These scripts are a convenient way to invoke CMake with the correct options.

Run the `rtime-make` script with the following command:

```
RTIMEHOME/resource/scripts/rtime-make --config Debug --target armv8leElfqcc8.3.0CERT-
↪QOS2.2.1 \
                      -G "Unix Makefiles" --build
```

Here is an explanation of each argument in the above command:

- `--config Debug`: Create Debug build.
- `--target <target>`: The target for the sources to be built. Refer to *Supported Platforms and Programming Languages* for the architecture abbreviations of supported platforms.
- `--build Build`: The generated project files.

To get a list of all the options, run:

```
rtime-make -h
```

To get help for a specific target, run:

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

### 3.2.2 Building the PSL with CMake

#### Preparing to build

RTI recommends creating a unique directory for each build configuration. A build configuration can be created to address specific architectures, compiler settings, or different *Connext Cert* build options.

RTI recommends assigning a descriptive *name* to each build configuration, using a common format. While there are no requirements to the format for functional correctness, the toolchain files in *Connext Cert* use the **RTIME_TARGET_NAME** variable to determine various compiler options and selections.

RTI uses the following format for the target architecture PSL:

```
{cpu}{compiler}{profile}-{OS}
```

- `{cpu}`: the CPU that the library was compiled for.
- `{compiler}`: the compiler used to build the library.
- `{profile}`: CERT if the library was built to be Cert-compatible; otherwise empty.
- `{OS}`: The operating system that the PSL was compiled for.

For example, the target name `armv8leElfqcc8.3.0CERT-QOS2.2.1` describes a PSL for *Connext Cert* for an Armv8 CPU, running QOS 2.2.1, compiled with gcc 8.3.0.

Files built by each build configuration will be stored under `RTIMEHOME/build/[Debug | Release]/<name>`. These directories are referred to as build directories or `RTIMEBUILD`. The structure of the `RTIMEBUILD` depends on the generated build files and should be regarded as an intermediate directory.

#### Creating build files from the command line

Open a terminal window in the `RTIMEHOME` directory and create the `RTIMEBUILD` directory. Change to the `RTIMEBUILD` directory and invoke CMake with the following arguments:

```
cmake -G <generator> -DCMAKE_BUILD_TYPE=<Debug | Release> \
      -DCMAKE_TOOLCHAIN_FILE=<toolchain file>  \
      -DRTIME_TARGET_NAME=<target-name>
```

Depending on the generator, do one of the following:

- For IDE generators (such as Eclipse, Visual Studio, Xcode), open the generated solution/project files and build the project/solution.
- For command-line tools (such as make, nmake, ninja), run the build-tool.

After a successful build, the output is placed in `RTIMEHOME/lib/<name>`.

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

- `ALL_BUILD`: Builds all the projects.
- `\rti_me_<name>`: Builds only the specific library. Note that that dependent libraries are built first.
- `ZERO_CHECK`: Runs CMake to regenerate project files in case something changed in the build input. This target does not need to be built manually.

For command-line tools, try `<tool> help` for a list of available targets to build. For example, if UNIX makefiles were generated:

```
make help
```

#### CMake flags used by Connext Cert

The following CMake flags (`-D`) are understood by *Connext Cert* and may be useful when building outside of the source bundle installed by RTI. An example would be incorporating the *Connext Cert* source in a project tree and invoking cmake directly on the `CMakeLists.txt` file provided by *Connext Cert*.

- `-DRTIME_TARGET_NAME=<name> -` The name of the target (equivalant to `--target` 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 Cert* 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 CMake will try to detect the system to build for.
- `-DRTI_NO_SHARED_LIB=true` - Do not build shared libraries. The default is undefined, which means shared libraries are built. NOTE: This flag must be undefined to build shared libraries. Setting the value to false is not supported.
- `-DRTI_MANUAL_BUILDID=true` - Do not automatically generate a build ID. The default value is undefined, which means CMake will generate a new build each time the libraries are built. Setting the value to false is not supported. The build ID is in its own source and only forces a recompile of a few files. Note that it is necessary to generate a build ID at least once (this is done automatically). Also, a build ID is not supported for cmake versions less than 2.8.11 because the TIMESTAMP function does not exist.
- `-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 a finite lease duration.

## 3.3 Compile-time options

The *Connext Cert* PSL source supports compile-time options. These options are generally used to control:

- Enabling/disabling features.
- Inclusion/exclusion of debug information.
- Inclusion/exclusion of APIs.
- Target platform definitions.
- Target compiler definitions.

**Note:** It is no longer possible to build a single library using CMake. Refer to *Custom Build Environments* for information on customized builds.

### 3.3.1 Debug information

Please note that *Connext Cert* debug information is independent of a debug build as defined by a compiler. In the context of *Connext Cert*, debug information refers to inclusion of:

- Logging of error-codes.
- Tracing of events.
- Precondition checks (argument checking for API functions).

Unless explicitly included/excluded, the following rule is used:

- For CMAKE_BUILD_TYPE = Release, the NDEBUG preprocessor directive is defined. Defining NDEBUG includes logging, but excludes tracing and precondition checks.
- For CMAKE_BUILD_TYPE = Debug, the NDEBUG preprocessor directive is undefined. With NDEBUG undefined, logging, tracing and precondition checks are included.

To manually determine the level of debug information, the following options are available:

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

### 3.3.2 Platform selection

The *Connext Cert* build system looks for target platform files in `RTIMEHOME/include/osapi`. All files that match `osapi_os_*.h` are listed under **RTIME_OSAPI_PLATFORM**. Thus, if RTI ports *Connext Cert* to a new platform, the target platform files 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)
- QNX (posix)

However, for custom ports, `<auto detect>` may not work. In that case, select the appropriate platform definition file instead of `<auto detect>`.

### 3.3.3 Compiler selection

The *Connext Cert* build system looks for target compiler files in `RTIMEHOME/include/osapi`. All files that match `osapi_cc_*.h` are listed under **RTIME_OSAPI_COMPILER**. Thus, if RTI adds a new compiler definition file, 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, `<auto detect>` may not work. In that case, select the appropriate compiler definition file instead of `<auto detect>`.

### 3.3.4 UDP options

Checking the **RTIME_UDP_ENABLE_IPALIASES** option 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** option enables UDP transformations in the UDP transport.

Checking the **RTIME_UDP_EXCLUDE_BUILTIN** option excludes the UDP transport from being built.

## 3.4 Custom build environments

The preferred method to build *Connext Cert* is to use CMake. However, in some cases it may be more convenient, or even necessary, to use a custom build environment. For example:

- Embedded systems often have numerous compiler, linker, and board-specific options that are easier to manage in a managed build.
- The compiler cannot be invoked outside of the build environment because it is 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 Cert*.

### 3.4.1 Importing the Connext Cert code

The process for importing the *Connext Cert* 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 Cert* source tree from the file-system. Note that some environments let you choose whether to make a copy-only link to the original files.
- Add the following include paths:
  - <root>/include/rti_me
  - <root>/include/rti_me/rti_me_psl
  - <root>/src/dds_c/domain
  - <root>/src/dds_c/infrastructure
  - <root>/src/dds_c/publication
  - <root>/src/dds_c/subscription
  - <root>/src/dds_c/topic
  - <root>/src/dds_c/type
- Add a compile-time definition `-DRTIME_TARGET_NAME="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"` .

# Chapter 4

# Platform Notes

## 4.1 Introduction

This section provides platform-specific instructions that you will need to build and run *RTI Connext Cert* applications.

For each supported operating system (OS), this section describes:

- Supported combinations of OS versions, CPUs, and compilers
- How to build your application, including:
  - Required *Connext Cert* and system libraries
  - Required compiler and linker flags
  - Details on how the *Connext Cert* libraries were built

To see a list of all supported platforms, refer to *Supported Platforms and Programming Languages*.

### 4.1.1 Library types

This section references Platform Independent Libraries (PIL) and Platform Support Libraries (PSL). These are library types that RTI provides in a *Connext Cert* installation. For more information, see *Library types*.

### 4.1.2 Supported libraries by platform

The following table shows which *Connext Cert* libraries are supported on each platform (RTI architecture).

Table 4.1: Supported Libraries by Platform

| Platform | RTI Architecture | Supported Libraries |
|---|---|---|
| QOS 2.2.1 (QNX OS for Safety) | armv8leElfqcc8.3.0CERT | `rti_me` `rti_me_netiozcopy` |

### 4.1.3 Supported transports by platform

The following table shows which transports are supported on each architecture.

Table 4.2: Supported Transports by Platform

| Platform | RTI Architecture | Intra | UDPv4 | Zero Copy v2 |
|---|---|---|---|---|
| QOS 2.2.1 (QNX OS for Safety) | armv8leElfqcc8.3.0CERT | ✓ | ✓ | ✓ |

## 4.2 QNX Platforms

The following table shows the currently supported QNX platforms.

Table 4.3: Supported QNX Platforms

| OS | Version | CPU | Network Stack | Toolchain | Architecture PIL | Architecture PSL |
|---|---|---|---|---|---|---|
| QOS (QNX OS for Safety) | 2.2.1 | ARMv8 (64-bit) | OS Default: io-pkt | qcc_gpp8.3.0 | armv8leElfqcc8.3.0CERT | armv8leElfqcc8.3.0CERT-QOS2.2.1 |

### 4.2.1 How the PIL was built for QNX platforms

This section describes how RTI built the Platform Independent Library (PIL) for QNX platforms.

The following table shows the compiler flags RTI used to create the PIL for QNX platforms:

Table 4.4: PIL Compiler Flags for QNX Platforms

| Architecture PIL | Library Format | Compiler Flags Used by RTI |
|---|---|---|
| armv8leElfqcc8.3.0CERT | Static Release | -Vgcc/8.3.0,gcc_ntoaarch64le -std=c99 -Winit-self -fstrict-aliasing -Wmissing-declarations -Wall -Wextra -Wpedantic -Wshadow -Wcast-align -Wunused -Wconversion -Wsign-conversion -Wlogical-op -Wdouble-promotion -DNDEBUG |

### 4.2.2 Building the PSL from source for QNX platforms

Refer to *Building the PSL* for instructions on how to build your own Platform Support Library (PSL) for QNX platforms.

### 4.2.3 Building QNX applications with Connext Cert

This section describes how RTI built the Platform Support Library (PSL) for QNX platforms. You must build applications with the same flags as the PSL in order to operate with *Connext Cert*. The PSL must also be binary compatible with the PIL.

The following table shows the compiler flags and required options that RTI used to build the PSL for QNX platforms. The PSL always requires the same compiler flags as the corresponding PIL, plus additional flags; the table below shows all these required flags.

When you build the PSL with `rtime-make`, the `--target` argument automatically adds all the necessary flags for the specified architecture.

Table 4.5: PSL Compiler Flags for QNX Platforms

| Architecture PSL | Library Format | Compiler Flags Used by RTI |
|---|---|---|
| armv8leElfqcc8.3.0CERT-QOS2.2.1 | Static Release | -Vgcc/8.3.0,gcc_ntoaarch64le -std=c99 -Winit-self -fstrict-aliasing -Wmissing-declarations -Wall -Wextra -Wpedantic -Wshadow -Wcast-align -Wunused -Wconversion -Wsign-conversion -Wlogical-op -Wdouble-promotion -DNDEBUG -DRTI_CERT -DRTI_PSL -DOSAPI_CC_DEF_H=<name file with compiler definitions> -DRTI_LITTLE_ENDIAN -DNDEBUG -I<install directory>/rti_me_cert.1.0/include/rti_me |

### 4.2.4 Allocating QNX resources

The following table outlines the type and amount of typical OS resources used by different entities and objects, allocated from the PSL:

| Entity | mutex | semaphore | timer |
|---|---|---|---|
| DDS_DomainParticipantFactory | 3 | 0 | 0 |
| DDS_DomainParticipant | 2 | 0 | 1 |
| DDS_DataReader | 1 | 0 | 0 |
| DDS_DataWriter | 1 | 0 | 0 |
| DDS_Publisher | 1 | 0 | 0 |
| DDS_Subscriber | 1 | 0 | 0 |
| DDS_WaitSet | 3 | 2 | 0 |
| DDS_GuardCondition | 1 | 0 | 0 |

# Chapter 5

# Getting Started

This section will help you get started with *Connext Cert* by using an example application that is available on GitHub. This is a HelloWorld example that can be altered to suit your preferences.

Additional examples can be generated with the included *rtiddsgen* tool to further explore *Connext Cert*'s features.

Once you have used the examples in this chapter to familiarize yourself with *Connext Cert*'s features, you are ready to develop your own applications as described in *Developing Applications*.

## 5.1 GitHub Example

*Connext Cert* provides a buildable example application on GitHub. It comes with instructions on how to build and run an application that performs basic publish-subscribe communication using Zero Copy transfer. It is only available in C, because *Connext Cert* does not include a C++ API.

Note that by default, the example links against release libraries. Refer to *Building Connext Cert* for instructions on how to build the *Connext Cert* release libraries.

## 5.2 Generating Examples

The RTI *Code Generator* (also referred to as *rtiddsgen*) included with *Connext Cert* can generate DDS example applications with a type definition file as input.

---

**Note:** Before running *rtiddsgen*, you might need to add `rti_connext_dds-<version>/rtiddsgen/scripts` to your path environment variable folder.

---

### 5.2.1 Default example

To generate an example, run the following command:

```
rtiddsgen -example -language <C> <file with type definition>
```

This generates an example using the default example template, which uses the Dynamic Participant Static Endpoint (DPSE) discovery plugin and UDP communication.

*rtiddsgen* accepts the following options:

- `-example`: Generates type files, example files, and CMakelists files.
- `-language <C>`: Generates C code.

The generated example can then be compiled using CMake and the `CMakelists.txt` file generated by *Code Generator*. *Code Generator* also creates a `README.txt` file with a description of the example and instructions for how to compile and run it.

### 5.2.2 Custom example

*Code Generator* can also generate examples using custom templates with the option `-exampleTemplate <templateName>`.

To generate an example using a custom template instead of the default one, run the following command:

```
rtiddsgen -example -exampleTemplate <template name> -language <C> <file with type␣
↪definition>
```

To see the list of the available templates for each language, run the following command:

```
rtiddsgen -showTemplates
```

As an example, the following command will generate an example in the C language, using the `zerocopy` custom template instead of the default template:

```
rtiddsgen -example -exampleTemplate zerocopy -language C <file with type definition>
```

### 5.2.3 Descriptions of generated examples

Each example consists of a publication and subscription pair to send and receive the type specified by the user. When compiled, the example creates two applications: one to send samples (a publisher) and another to receive samples (a subscriber).

- **default example (no template specified)**

  Discovery of endpoints is done with the static-endpoint discovery (DPSE). Only the UDP and INTRA transports are enabled. The subscriber application creates a *DataReader*, which uses a listener to receive notifications about new samples and matched publishers. These notifications are received in the middleware thread (instead of the application thread).

This example uses a static UDP interface configuration. Using this API, the UDP transport is statically configured. This is useful in systems that are not able to return the installed UDP interfaces (name, IP address, mask, etc.).

- **zerocopy**

  Identical to the default template, except that the only transport used is Zero Copy v2. The UDP transport remains enabled because it is used for DDS discovery.

  Both the publisher and subscriber applications must run in the same OS instance.

### 5.2.4 How to compile the generated examples

Before compiling, set the environment variable `RTIMEHOME` to the *Connext Cert* installation directory.

The *Connext Cert* source bundle includes rtime-make to simplify invoking CMake. This script is a convenient way to invoke CMake with the correct options. For example:

```
cd <directory with generated example>

rtime-make --config <Debug|Release> --build --target armv8leElfgcc7.3.0-Linux4 --source-
↪dir . \
     -G "Unix Makefiles" --delete [-DRTIME_IDL_ADD_REGENERATE_TYPESUPPORT_RULE_eq_true]
```

**Warning:** RTI recommends using the toolchain file that matches the target architecture to compile the generated examples.

For example, if the target architecture is `--target armv8leElfgcc7.3.0-Linux4`, then the example applications should be compiled with the `armv8leElfgcc7.3.0-Linux4` toolchain file. Failing to do so may cause warnings.

The executable can be found in the `objs` directory.

The following options are accepted:

- `-DRTIME_IDL_ADD_REGENERATE_TYPESUPPORT_RULE=true` adds a rule to regenerate type support plugin source files if the input file with the type definition changes. Default value is 'false'.

### 5.2.5 How to run the generated examples

**Note:** The examples are hard coded to use the loopback address 127.0.0.1.

Run the subscriber with this command:

```
objs/armv8leElfgcc7.3.0-Linux4/<Type definition file name>_subscriber [-domain
↪<Domain_ID>] [-peer <address>] \
             [-sleep <sleep_time>] [-count <seconds_to_run>]
```

and run the publisher with this command:

```
objs/armv8leElfgcc7.3.0-Linux4/<Type definition file name>_publisher [-domain
↪<Domain_ID> -peer <address>] \
             [-sleep <sleep_time>] [-count <seconds_to_run>]
```

# Chapter 6

# Developing Applications

This section describes how to write *Connext Cert* applications. It covers preparing your development environment, defining data types, generating support code for your data types, and creating the entities that publish and subscribe to data.

For a deeper dive into *Connext Cert*'s features, refer to the *User's Manual*.

## 6.1 Prepare Your Development Environment

This section describes how to set up your development environment for *Connext Cert* applications, such as the required environment variables, compilers, compiler definitions, and libraries.

### 6.1.1 Set environment variables

The `RTIMEHOME` environment variable must be set to the installation directory path for *RTI Connext Cert*. If you installed *RTI Connext* with default settings, *RTI Connext Cert* will be here: `<path_to_connext_dds_installation>/rti_connext_dds-<version>/rti_connext_micro-<version>`. If you copied *RTI Connext Cert* to another place, set `RTIMEHOME` to point to that location.

### 6.1.2 Add required preprocessor flags

All *Connext Cert* applications require the following preprocessor defines:

```
-DRTI_PSL=1
-IRTIMEHOME/include
-IRTIMEHOME/include/rti_me
-IRTIMEHOME/include/rti_me/rti_me_psl
```

Add the following preprocessor defines, according to your platform and compiler:

Using QCC:

```
-DOSAPI_CC_DEF_H=osapi/osapi_cc_qcc.h
-DRTI_QNX
```

### 6.1.3 Link applications and libraries

Add the library path for both the PIL and PSL to the linker's search path:

- `RTIMEHOME/lib/<arch>/` (PIL)
- `RTIMEHOME/lib/<arch>-<PSL>/` (PSL)

---

**Note:** When executing executables that are linked with the *Connext Cert* shared libraries, you must add the path to the PIL architecture directory to the runtime linker's search path.

---

To link a C application, the libraries are required in the following order:

- `RTIMEHOME/lib/<arch>/`

1. `rti_me` (always required)

- `RTIMEHOME/lib/<arch>-<PSL>/`

    2. `rti_me_psl` (when building with a Platform Independent Library)
    3. `rti_me_netiozcopy` (when building with the Zero Copy v2 transport)

## 6.2 Define a Data Type

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

*Connext Cert* accepts types definitions in Interface Definition Language (IDL) format.

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

```
::

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

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

## 6.3 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_me_cert.1.0/rtiddsgen/scripts*.

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

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

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

- The `-micro` option is necessary to generate support code specific to *Connext Cert*, as *rtiddsgen* is also capable of generating support code for *Connext*, and the generated code for the two are different. Note that RTI *Connext Micro* and *Connext Cert* use the same *rtiddsgen* and similar code is generated. However, please note that the generated code *must* be compiled with the RTI_CERT flag when linking against the *Connext Cert* libraries.
- The `-language` option specifies the language of the generated code. *Connext Cert* supports only C
- The `-replace` option specifies that the new generated code will replace, or overwrite, any existing files with the same name.

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

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

## 6.4 Create an Application

The rest of this guide will walk you through the steps to create 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*).

### 6.4.1 Registry Configuration

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

The *Connext Cert* architecture 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 (only available in *Connext Micro*) with an RT component factory. Each RT component factory must be registered within an RT registry in order for its components to be usable by an application.

*Connext Cert* automatically registers components that provide necessary functionality. These include components for DDS *Writers* and *Readers*, the 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 the name “wh”.
- **Reader History**. Queue of received samples of a *DataReader*. Must be registered with the name “rh”.
- **Discovery (DPSE)**. Discovery component with 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",
                          RHSM_HistoryFactory_get_interface(),
                          NULL, NULL))
{
    /* failure */
}
```

Only one discovery component can be registered. Each has its own properties that can be configured upon registration. Please note that only DPSE can be registered in *Connext Cert*. Each has its own properties that can be configured upon registration.

- 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;
}
```

## 6.5 Configure UDP Transport

You will need to configure and register the UDP transport component with *Connext Cert*. To do this, create an instance of the UDP properties, add one or more network interfaces to the properties, then register the UDP transport.

Example code:

- Create an instance of the UDP transport properties:

```
#include "netio/netio_psl_udp.h"

UDPv4_TransportProperty_T *udp_property = NULL;
udp_property = UDPv4_TransportProperty_new();
```

- Add a network interface to the UDP transport properties:

```
/* This function takes the following arguments:
 * Param 1 is the UDP property
 * Param 2 is the IP address of the interface in host order
 * Param 3 is the Netmask of the interface
 * Param 4 is the name of the interface
 * Param 5 are flags. The following flags are supported (use OR for multiple):
 *      UDP_INTERFACE_INTERFACE_UP_FLAG - Interface is up
 *      UDP_INTERFACE_INTERFACE_MULTICAST_FLAG - Interface supports multicast
 */
if (!UDPv4_InterfaceTable_add_entry(
            udp_property,
            0x7f000001, /* 127.0.0.1 */
            0xff000000,
            "lo",
            UDP_INTERFACE_INTERFACE_UP_FLAG |
            UDP_INTERFACE_INTERFACE_MULTICAST_FLAG))
{
    printf("ERROR: Failed to add interface\n");
}
```

- Register the UDP component with updated properties:

```
const char* const MY_UDP_NAME = "_udp";

if (!UDPv4_Interface_register(registry, MY_UDP_NAME, udp_property))
{
    /* failure */
}
```

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

## 6.6 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 (typically "dpde" or "dpse") must be assigned to the DiscoveryQosPolicy's name. Please note that in *Connext Cert*, only the DPSE discovery plugin is supported.
- **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 */
```

(continues on next page)

(continued from previous page)

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

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

### 6.6.1 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 */
}
```

### 6.6.2 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)
{
```

(continues on next page)

(continued from previous page)

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

### 6.6.3 DPSE Discovery: Assert Remote Participant

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

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

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

## 6.7 Create Publisher

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

In *Connext Cert*, 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.

## 6.8 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;
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");
    }
}
```

### 6.8.1 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 */
}
```

### 6.8.2 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!");
```

(continues on next page)

(continued from previous page)

```
/* 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*.

## 6.9 Create Subscriber

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

In *Connext Cert*, 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.

## 6.10 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 =
```

(continues on next page)

(continued from previous page)

```
    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)
{
    /* 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;
```

(continues on next page)

(continued from previous page)

```
    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)
    {
        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);
}
```

### 6.10.1 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 A PI 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.

### 6.10.2 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 */
}
```

(continues on next page)

(continued from previous page)

```
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 */
}

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 ==
```

(continues on next page)

(continued from previous page)

```
                    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;
    }
}
```

### 6.10.3 Filtering Samples

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

# Chapter 7

# User's Manual

## 7.1 Initializing the Connext Cert Library

*Connext Cert* has been designed to integrate with a wide range of operating systems, network stacks, and CPUs. For this reason, *Connext Cert* places few restrictions on how it is integrated. The memory management API defined by *Connext Cert* may be implemented using standard C libray APIs such as *malloc()* and *free()*, or something hardware specific relying on memory being allocated from a specific memory region.

In order to allow a degree of flexibility, integrations may be configurable at run-time. This configuration may require validation before it is safe to make specific calls, such as allocating memory.

In order to guarantee consistency accross all integrations for when it is safe to call APIs, *Connext Cert* requires that its library is initialized with *DDS_DomainParticipantFactory_get_instance* before any public APIs are called, unless an API is documented to be safe to call before *DDS_DomainParticipantFactory_get_instance*. *DDS_DomainParticipantFactory_get_instance* initializes an integration, providing an opportunity for integrations to validate its configuration.

---

**Note:** This restriction is not limited to DDS APIs, but extends to **all** public APIs, such as sequence APIs, type-support APIs, string APIs, and component APIs.

---

*Connext Cert* is initialized with a successful call to *DDS_DomainParticipantFactory_get_instance*. On success, *DDS_DomainParticipantFactory_get_instance* returns a reference to a *DDS_DomainParticipantFactory*; on failure, 'nil' is returned:

```
DDS_DomainParticipantFactory *factory = NULL;

factory = DDS_DomainParticipantFactory_get_instance();

if (factory == NULL)
{
```

(continues on next page)

(continued from previous page)

```
    /* something failed, exit */
    exit(-1);
}

/* Safe to call other public APIs */
```

After a successful call to *DDS_DomainParticipantFactory_get_instance*, public APIs are safe to call as documented. APIs that **must not** be called before *DDS_DomainParticipantFactory_get_instance* have the following additional description:

```
API Restriction:
This function must only be called after DDS_DomainParticipantFactory_get_instance.
```

**Warning:** *DDS_DomainParticipantFactory_get_instance* is not guarenteed to be thread-safe.

### 7.1.1 rtiddsgen

*rtiddsgen* is the type support compiler included with *Connext Cert*. *rtiddsgen* generates code to send and receive data types across the network, as well as to allocate memory to store data types in memory. These memory allocations use the memory management APIs defined by *Connext Cert*.

Because each integration determines how these APIs are implemented, it is important that Type-Support APIs are **not** called until *Connext Cert* has been initialized. APIs such as *FooTypeSupport_create_data* and *FooTypeSupport_delete_data* are **not** safe to call until after a successful call to *DDS_DomainParticipantFactory_get_instance*:

```
DDS_DomainParticipantFactory *factory = NULL;
Foo *sample = NULL;

/* NOT ALLOWED */
sample = FooTypeSupport_create_data();

factory = DDS_DomainParticipantFactory_get_instance();

if (factory == NULL)
{
    /* something failed, exit */
    exit(-1);
}

/* ALLOWED */
sample = FooTypeSupport_create_data();
if (sample == NULL)
{
    /* something failed, exit */
    exit(-1);
}
```

(continues on next page)

(continued from previous page)

```
/* Calls other public APIs */
```

### 7.1.2 The Connext Cert System API

The *Connext Cert* System API enables applications to configure the behavior of *Connext Cert* at runtime. Which configuration options are available depends on the specific integration.

However, because the *Connext Cert* system must be configured **before** *Connext Cert* is initialized, it is safe to call public System APIs before *DDS_DomainParticipantFactory_get_instance*, such as *OSAPI_System_get_property* and *OSAPI_System_set_property*:

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

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

/* Set sys_property */

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

factory = DDS_DomainParticipantFactory_get_instance();
if (factory == NULL)
{
    /* error */
    return;
}
```

### 7.1.3 Component Registration

*Connext Cert* consists of core APIs and additional components that extend its functionality. *Connext Cert* includes two components which **must** always be registered with *Connext Cert* before any DDS entities can be created: the *writer* and *reader* history caches. The following code sample demonstrates how to register these:

```
#include "wh_sm/wh_sm_history.h"
#include "rh_sm/rh_sm_history.h"



DDS_DomainParticipantFactory *factory = NULL;
```

(continues on next page)

(continued from previous page)

```
RT_Registry_T *registry = NULL;

factory = DDS_DomainParticipantFactory_get_instance();

if (factory == NULL)
{
    /* something failed, exit */
    exit(-1);
}

registry = DDS_DomainParticipantFactory_get_registry(factory);

if (registry == NULL)
{
    /* something failed, exit */
    exit(-1);
}

if (!RT_Registry_register(registry, DDSHST_WRITER_DEFAULT_HISTORY_NAME,
                          WHSM_HistoryFactory_get_interface(), NULL, NULL))
{
    /* something failed, exit */
    exit(-1);
}

if (!RT_Registry_register(registry, DDSHST_READER_DEFAULT_HISTORY_NAME,
                          RHSM_HistoryFactory_get_interface(), NULL, NULL))
{
    /* something failed, exit */
    exit(-1);
}
```

*Connext Cert* includes other components, such as *Discovery* plugins. These are documented in other sections.

## 7.2 Data Types

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

*Connext Cert* data samples sent on the same *Connext Cert* 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 Cert*, 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 (only in *Connext Micro* DDS data samples and how to get and set their fields. These tasks are described in the section on Working with DDS Data Samples in the RTI Connext DDS Core Libraries User's Manual (available here if you have Internet access).

### 7.2.1 Introduction to the Type System

A *user data type* is any custom type that your application defines for use with *RTI Connext Cert*. 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 Cert*; 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 Cert*, you must define that type in a way the middleware understands and then register the type with the middleware. These steps allow *Connext Cert* 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 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 Cert* 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.

The bound is either *excplict* or *implicit*:

1. An *explicit* bound is given directly in the IDL:

```
struct MyType
{
    //Maximum of 32 longs
    sequence<32> a_long_seq;
}
```

2. An *implicit* bound uses the unbounded notation in IDL, but relies on the -sequenceSize parameter passed to *rtiddsgen* for the maximum length:

```
struct MyType
{
    sequence<long> a_long_seq;
}

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

### Strings and Wide Strings

*Connext Cert* 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 Cert* are large enough to store 4-byte Unicode/UTF16 characters.

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

In C and Traditional C++, strings are mapped to *char**.

The bound is either *excplict* or *implicit*:

1. An *explicit* bound is given directly in the IDL:

```
struct MyType
{
    //Maximum of 32 bytes + NUL termination
    string<32> a_string;
}
```

2. An *implicit* bound uses the unbounded notation in IDL, but relies on the -stringSize parameter passed to *rtiddsgen* for the maximum length:

```
struct MyType
{
    // Unbounded notation, but not unbounded. Bound determined
    // by the -stringSize parameter to rtiddsgen
    string a_string;
}


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

### IDL String Encoding

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

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

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

- C, Traditional C++ (only in *Connext Micro*)

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

#### IDL Wide Strings Encoding

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

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

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

- C, Traditional C++

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

#### Sending Type Information on the Network

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

### 7.2.2 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++ (only *Connext Micro*). *RTI Code Generator* parses the IDL file and automatically generates all the necessary routines and wrapper functions to bind the types for use by *Connext Cert* at run time. You will end up with a set of required routines and structures that your application and *Connext Cert* will use to manipulate the data.

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

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

### 7.2.3 Working with DDS Data Samples

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

**In C:**

You create and delete your own objects from factories, just as you create *Connext Cert* 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*/

/* not support in Micro Cert */
MyDataTypeSupport_delete_data(sample);
```

## 7.3 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.

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

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

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

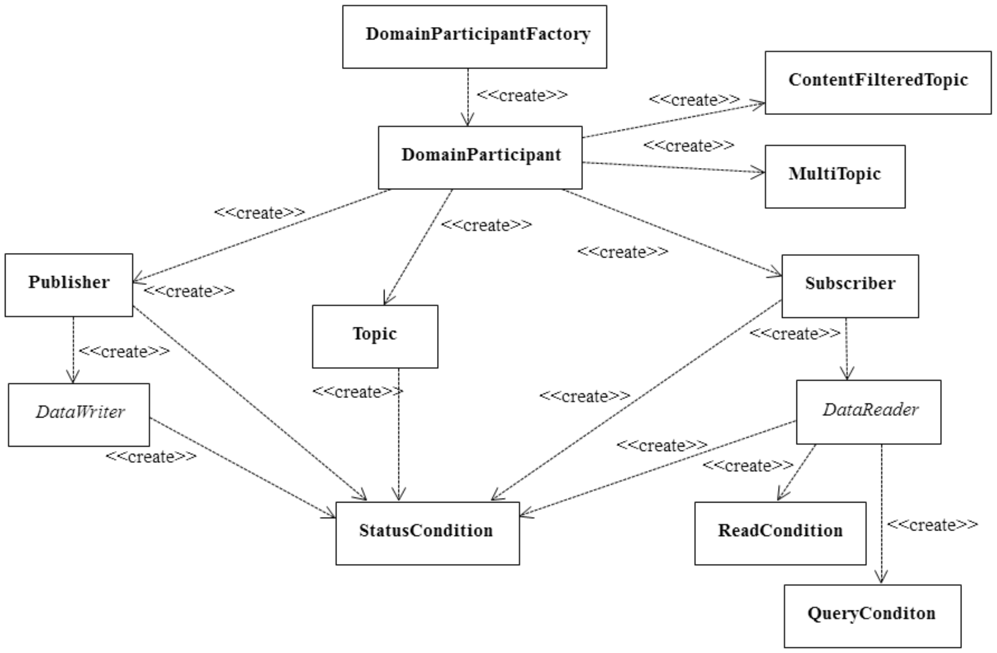


Figure 7.1: Overview of DDS Entities

## 7.4 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 documentation.

### 7.4.1 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.

### 7.4.2 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*.

### 7.4.3 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 documentation.

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

### 7.4.4 Publisher QosPolicies

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

### 7.4.5 DataWriter QosPolicies

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

## 7.5 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 documentation.

### 7.5.1 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 Cert* 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 Cert* 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 Cert* 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 Cert* 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 Cert* 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*.

### 7.5.2 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 Cert* notifies your application of status changes relevant to the *Entity*. The QosPolicies allow your application to configure the behavior and resources of the *Entity*.

- The *DomainParticipant* defines the DDS domain on which the information will be available.
- The *Topic* defines the name of the data to be subscribed, as well as the type (format) of the data itself.

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

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

### 7.5.3 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 HTML documentation.

### 7.5.4 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*.

### 7.5.5 Subscriber QosPolicies

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

### 7.5.6 DataReader QosPolicies

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

## 7.6 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 Cert* application. For specific details on any mentioned operations, see the C API Reference documentation.

### 7.6.1 Fundamentals of DDS Domains and DomainParticipants

*DomainParticipants* are the focal point for creating, destroying (only in *Connext Micro*), and managing other *RTI Connext Cert* objects. A DDS *domain* is a logical network of applications: only applications that belong to the same DDS *domain* may communicate using *Connext Cert*. 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.

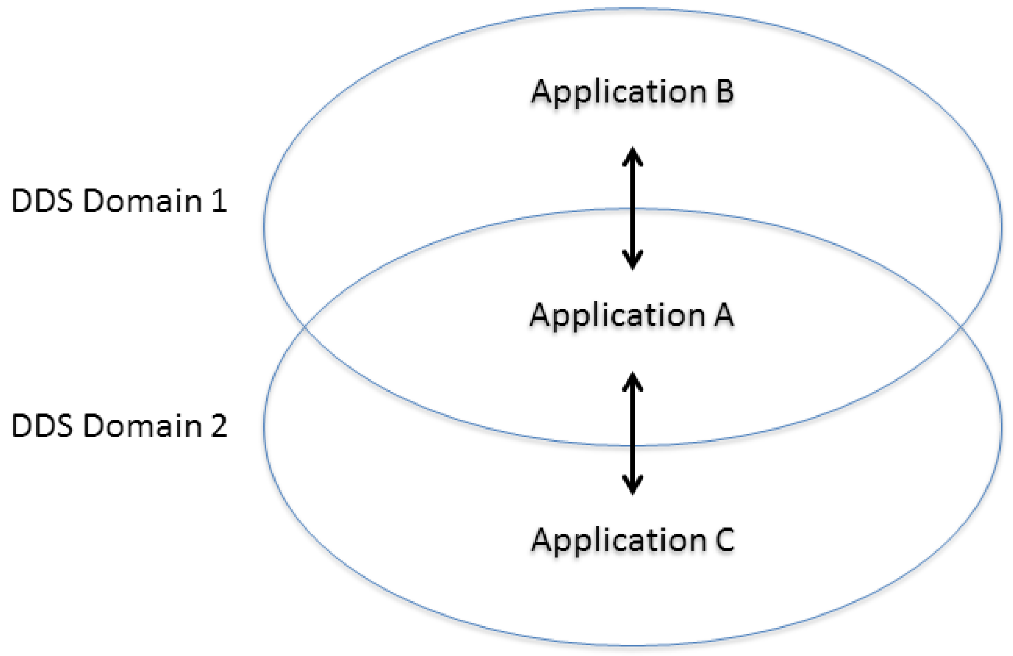


Figure 7.2: Relationship between Applications and DDS Domains
Applications can belong to multiple DDS domains—*A* belongs to DDS domains 1 and 2. Applications in the same DDS domain can communicate with each other, such as *A* and *B*, or *A* and *C*. Applications in different DDS domains, such as *B* 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*).

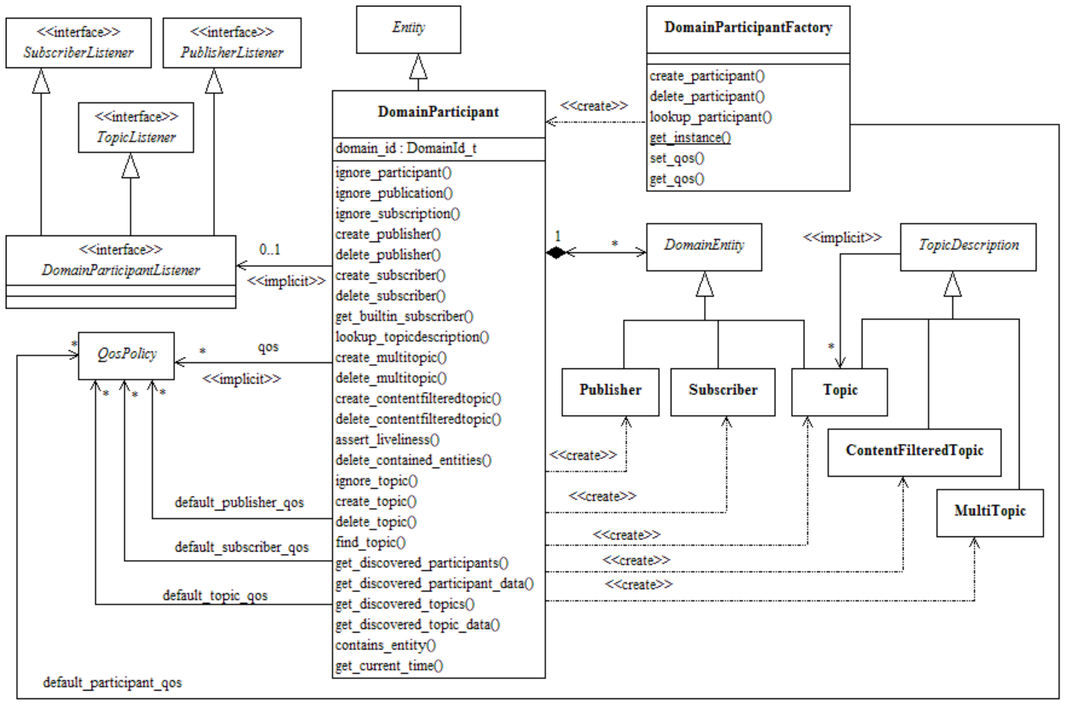


Figure 7.3: DDS Domain Module
Note: MultiTopics are not supported.

## 7.6.2 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.

## 7.7 Transports

### 7.7.1 Introduction

In *RTI Connext Cert*, DDS entities exchange information using transports. Transports exhange data with peer transports, and *Connext Cert* entities can generally exchange information using different types of transports, e.g. UDPv4 or a serial port. All transports send and receive RTPS messages encapsulated in the transport's native format, e.g. UDP packets.

**Note:** This version of *Connext Cert* only supports UDPv4 and a special transport for internal communication within a DDS *DomainParticipant*.

*Connext Cert* has a pluggable-transport architecture. The core of *Connext Cert* is transport agnostic; it does not make any assumptions about the actual transports used to send and receive messages. Instead, *Connext Cert* 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.

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

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

Note that *Connext Cert* does not support RTPS fragmentation and is limited to IDL data types less than or equal to 63000 bytes **or** the maximum transmission unit (MTU) of the underlying transport, whichever is smaller.

Also note that *Connext Cert* does not query the MTU size from the registered transport plugins. If an IDL data-type exceeds the MTU size, the data will be silently discarded.

*Connext Cert* does not track the maximum receive unit (MRU) of other nodes in the system. Therefore, *Connext Cert* relies on consistent configuration across all the nodes in the system in order to successfully send and receive data. For example, if a *Connext Cert* node has a MRU of 8000 bytes and another *Connext Cert* node sends 9000 bytes (with a sufficiently large MTU), the data will be sent, but not received.

### 7.7.2 Transport Limits

The following limitations apply to all *Connext Cert* transports.

#### IDL Data Types and Size

*Connext Cert* does not support RTPS fragmentation and is limited to IDL data types less than or equal to 63000 bytes *or* the maximum transmission unit (MTU) of the underlying transport, whichever is smaller.

#### Maximum Transmission Unit (MTU)

*Connext Cert* does not query the MTU size from the registered transport plugins. If the MTU size is exceeded, the data will be silently discarded.

### Maximum Receive Unit (MRU)

*Connext Cert* does not track the maximum receive unit (MRU) of other nodes in the system. Therefore, *Connext Cert* relies on consistent configuration accross all the nodes in the system in order to successfully send and receive data. For example, if a node has a MRU of 8000 bytes and another node sends 9000 bytes (with a sufficiently large MTU), the data will be sent, but not received.

## 7.7.3 Transport Registration

*RTI Connext Cert* supports different transports, and transports must be registered with *Connext Cert* before they can be used by an application.

Transports must be registered with a unique name; this name is later used to configure where to send and receive discovery and user data. A transport name cannot exceed seven (7) UTF-8 characters.

**Note:** The name the transport is registered with has no meaning to *Connext Cert*. A transport plugin may allow multiple registerations under different names. For example, two UDPv4 transport plugins may be registered, one for discovery and one for user data, with different properties.

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

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

```
DDS_DomainParticipantFactory *factory;
RT_Registry_T *registry;

/* In general properties must be available for the lifetime of the
 * transport.
 */
struct UDP_InterfaceFactoryProperty udp_property1;
struct UDP_InterfaceFactoryProperty udp_property2;

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

/* Set UDP properties */
if (!RT_Registry_register(registry,"myudp1",
                          UDP_InterfaceFactory_get_interface(),
                          &udp_property2._parent._parent,NULL))
{
    return error;
```

(continues on next page)

(continued from previous page)

```
}

/* Set UDP properties */
if (!RT_Registry_register(registry,"myudp2",
                          UDP_InterfaceFactory_get_interface(),
                          &udp_property2._parent._parent,NULL))
{
    return error;
}
```

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

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

To enable both transports:

```
DDS_StringSeq_set_maximum(&dp_qos.transports.enabled_transports,2);
DDS_StringSeq_set_length(&dp_qos.transports.enabled_transports,2);
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports,0) =
                                                 REDA_String_dup("myudp1");
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports,1) =
                                                 REDA_String_dup("myudp2");
```

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

### 7.7.4 Transport Addresses

In order for DDS entities to communicate, the DDS entities must know each other's location. DDS entities may be colocated in the same DDS *DomainParticipant*, may be located in different DDS *DomainParticipants* within the same node, or may be located on different nodes connected by a network.

In DDS, a location is called a *locator*. A locator uniquely describes how to reach one or more DDS entitites in a network. A DDS locator consists of the following parts:

- The locator *kind* identifies the type of locators, e.g. UDPv4.
- The locator *port* identifies the location of DDS entities at an address. 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.

- The locator *address* indentifies the network address. Transports are concerned with exchanging messages using the network address.

### Reserving Addresses and Ports

Address reservation is the process to determine which locators should be used in the discovery announcement. Which transports and addresses to be used are 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 releases the port on *all* transports and tries again. If no free port can be found, the process fails and the *DomainParticipant* cannot be created.

**Warning:** If an address is specified without the transport name as a prefix, e.g. "192.168.1.1" instead of "_udp://192.168.1.1", and multiple transports understand the address, only the *last* transport found will try to reserve the address. Which transport is the *last* is non-determinstic. This capability is present to be backwards compatible with earlier versions of *Connext Cert*, but **should not be used**; this feature may be deprecated in future versions. Always specify addresses using the transport name as the prefix.

### Address Limitations

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

### Address Notation

In *Connext Cert*, all addresses are specified as ASCII strings. The full address format is:

```
< > denotes optional
[ ] denotes range or discreet values, unless enclosed in ''
    which means a literal.

ADDRESS = <PREFIX://><ADDRESS> |
          @<PREFIX://><ADDRESS> |
          INDEX@<PREFIX>://<ADDRESS>

INDEX = INTEGER | '[' INTEGER ']' | '[' INTEGER-INTEGER ']' | '[' -INTEGER ']'

PREFIX = [a-zA-Z_][0-9a-zA-Z_]+

INTEGER = DEC_INTEGER | HEX_INTEGER

DEC_INTEGER = [0-9]+

HEX_INTEGER = [0x|0X][0-9a-fA-F]+

ADDRESS = 0 or more 8bit characters
```

Note that while the `PREFIX` is marked optional, it should always be used.

## 7.7.5 RTPS

The RTPS transport encapsulates user-data in RTPS messages and parses received RTPS messages for user-data. This chapter describes how to configure RTPS.

### Registration of RTPS

RTPS is automatically registered when a *DDS_DomainParticipantFactory* is initialized with *DDS_DomainParticipantFactory_get_instance()*. In order to change the RTPS configuration, it is necessary to first unregister it from the participant factory, set the properties, and then register RTPS with the new properties. This process is identical to other plugins in *Connext Cert*, such as the UDP transport and discovery plugins.

The following code shows the steps:

```
int main(int argc,char *argv)
{
    struct RTPS_InterfaceFactoryProperty *rtps_property = NULL;
    DDS_DomainParticipantFactory *factory = NULL;
    RT_Registry_T *registry = NULL;
    struct RTPS_InterfaceFactoryProperty *rtps_property = NULL;

```

(continues on next page)

(continued from previous page)

```
    /* get the Domain Participant factory and registry*/
    factory = DDS_DomainParticipantFactory_get_instance();

    registry =  DDS_DomainParticipantFactory_get_registry
                    (DDS_DomainParticipantFactory_get_instance());


    /* unregister the RTPS transport */
    if (!RT_Registry_unregister(registry, NETIO_DEFAULT_RTPS_NAME,
                                NULL,NULL))
    {
        printf("failed to unregister rtps\n");
        return 0;
    }

    rtps_property = (struct RTPS_InterfaceFactoryProperty *)
            malloc(sizeof(struct RTPS_InterfaceFactoryProperty));

    if (rtps_property == NULL)
    {
        printf("failed to allocate rtps properties\n");
        return 0;
    }

    /* Set the new properties and register RTPS again */

    if (!RT_Registry_register(registry, NETIO_DEFAULT_RTPS_NAME,
                RTPS_InterfaceFactory_get_interface(),
                (struct RT_ComponentFactoryProperty*)rtps_property,
                 NULL))
    {
        printf("failed to register rtps\n");
        return 0;
    }


    DDS_DomainParticipantFactory_create_participant(
       factory, domain_id,&dp_qos, NULL,DDS_STATUS_MASK_NONE);
}
```

Please note that the RTPS properties *must* be valid for the *entire* life-cycle of the participant factory because RTPS *does not* make an internal copy. This saves memory when properties are stored in preallocated memory (for example in ROM).

### Overriding the Builtin RTPS Checksum Functions

Some applications may require specialized functions to guarantee message integrity or may have special hardware that supports faster checksum calculations. *Connext Cert* provides a way for users to override the builtin checksum functions. Note that if a different checksum is calculated it may prevent interoperability with other DDS implementations.

### Checksum function definition

A checksum function must define a structure of the following type:

```
typedef struct RTPS_ChecksumClass
{
    RTPS_ChecksumClassId_T class_id;
    void *context;
    RTPS_CalculateChecksum_T calculate_checksum;
} RTPS_ChecksumClass_T;
```

The type has three members:

1. class_id - The class ID must be:
   - RTPS_CHECKSUM_CLASSID_BUILTIN32 for the 32-bit checksum.
   - RTPS_CHECKSUM_CLASSID_BUILTIN64 for the 64-bit checksum.
   - RTPS_CHECKSUM_CLASSID_BUILTIN128 for the 128-bit checksum.
2. context - An opaque object for you to provide context for this function. This context will be passed to the *calculate_checksum* every time it is called.
3. checksum_calculate - The function pointer to the checksum function. The function is defined as

   ```
   typedef RTI_BOOL
   (*RTPS_ChecksumCalculate_T)(void *context,
                               const struct REDA_Buffer *buf,
                               RTI_UINT32 buf_length,
                               RTPS_Checksum_T *checksum);
   ```

   - context: *Connext Cert* will pass in the context as defined in the class.
   - buf: An array of REDA_Buffer. Each REDA_Buffer includes a pointer and size of the buffer.
   - buf_length: The size of the array.

   RTPS_Checksum_T checksum: This is the out parameter of this function. It is a union defined as follows:

   ```
   typedef union RTPS_Checksum
   {
           RTI_UINT32 checksum32;
   ```

(continues on next page)

(continued from previous page)

```
        RTI_UINT64 checksum64;
        RTI_UINT8 checksum128[16];
} RTPS_Checksum_T;
```

Please note the following *important* information regarding the output values:

1. The number returned in checksum32 is assumed to be in *host order* endinaness.
2. The number returned in checksum64 is assumed to be in *host order* endinaness.
3. checksum128 is treated as an octet array.

### Example

Below is an example implementation of a custom CRC-32 function using the Intel intrinsic functions. It shows the QoS that needs to be set, as well as how to override the builtin checksum function.

```
RTI_BOOL
CrcClassTest_custom_crc32_other(void *context,
                                const struct REDA_Buffer *buf,
                                unsigned int buf_length,
                                union RTPS_CrcChecksum *checksum)
{
    RTI_UINT32 crc = 0;
    unsigned char *data = (unsigned char *) buf[0].pointer;
    RTI_UINT32 length = buf[0].length;
    int k;

    UNUSED_ARG(k);
    UNUSED_ARG(context);
    UNUSED_ARG(buf_length);

    for (k = 0; k < length; k++)
    {
        crc = _mm_crc32_u8(crc, data[k]);
    }

    checksum->checksum32 = crc;

    return RTI_TRUE;
}


int main(int argc,char *argv)
{
    struct DDS_DomainParticipantQos dp_qos =
            DDS_DomainParticipantQos_INITIALIZER;
    struct RTPS_InterfaceFactoryProperty *rtps_property = NULL;
    DDS_DomainParticipantFactory *factory = NULL;
    RT_Registry_T *registry = NULL;
    struct RTPS_InterfaceFactoryProperty *rtps_property = NULL;
```

(continues on next page)

(continued from previous page)

```
    /* Instantiate a RTPS_CrcClass for your custom function*/
    struct RTPS_ChecksumClass custom_crc32 =
    {
        RTPS_CHECKSUM_CLASSID_BUILTIN32, /*class_id*/
        NULL, /*context*/
        CrcClassTest_custom_crc32_other /*Custom function*/
    };

    /* get the Domain Participant factory and registry*/
    factory = DDS_DomainParticipantFactory_get_instance();

    registry =  DDS_DomainParticipantFactory_get_registry
                    (DDS_DomainParticipantFactory_get_instance());


    /* unregister the RTPS transport */
    if (!RT_Registry_unregister(registry, NETIO_DEFAULT_RTPS_NAME,
                                NULL,NULL))
    {
        printf("failed to unregister rtps\n");
        return 0;
    }

    rtps_property = (struct RTPS_InterfaceFactoryProperty *)
            malloc(sizeof(struct RTPS_InterfaceFactoryProperty));

    if (rtps_property == NULL)
    {
        printf("failed to allocate rtps properties\n");
        return 0;
    }


    /* the rtps property takes the structure with the custom
     * function
     */

    *rtps_property = RTPS_INTERFACE_FACTORY_DEFAULT;
    rtps_property->checksum.allow_builtin_override = RTI_TRUE;
    rtps_property->checksum.builtin_checksum32_class = custom_crc32;

    /* register the RTPS transport */
    if (!RT_Registry_register(registry, NETIO_DEFAULT_RTPS_NAME,
                RTPS_InterfaceFactory_get_interface(),
                (struct RT_ComponentFactoryProperty*)rtps_property,
                 NULL))
    {
        printf("failed to register rtps\n");
        return 0;
    }
```

(continues on next page)

(continued from previous page)

```
    /* modify the domain participant qos */
    dp_qos.protocol.compute_crc = DDS_BOOLEAN_TRUE;
    dp_qos.protocol.check_crc = DDS_BOOLEAN_TRUE;
    dp_qos.protocol.require_crc = DDS_BOOLEAN_TRUE;
    dp_qos.protocol.computed_crc_kind =  DDS_CHECKSUM_BUILTIN32;
    dp_qos.protocol.allowed_crc_mask = DDS_CHECKSUM_BUILTIN32;

    /* use the qos and the factory to create a participant */

    DDS_DomainParticipantFactory_create_participant(
       factory, domain_id,&dp_qos, NULL,DDS_STATUS_MASK_NONE);
}
```

### 7.7.6 INTRA Transport

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

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

#### Registering the INTRA Transport

The builtin INTRA transport is a *RTI Connext Cert* 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. */
```

(continues on next page)

(continued from previous page)

```
REDA_StringSeq_set_maximum(&dp_qos.user_traffic.enabled_transports,1);
REDA_StringSeq_set_length(&dp_qos.user_traffic.enabled_transports,1);
*REDA_StringSeq_get_reference(&dp_qos.user_traffic.enabled_transports,0) =
                                    REDA_String_dup("_udp://");
```

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

#### Reliability and Durability

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

#### Threading Model

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

This model has two *important limitations*:

- Because a *DataReader*'s on_data_available() listener is called in the context of the *DataWriter*'s send thread, a *DataReader* may potentially process data at a different priority than intended (the *DataWriter*'s). While it is generally not recommended to process data in a *DataReader*'s on_data_available() listener, it is particularly important *to not do so* when using the INTRA transport. Instead, use a DDS WaitSet or a similar construct to wake up a separate thread to process data.
- Because a *DataReader*'s on_data_available() listener is called in the context of the *DataWriter*'s send thread, any method called in the on_data_available() listener is done in the context of the *DataWriter*'s stack. Calling a *DataWriter* **write()** in the callback could result in an infinite call stack. Thus, it is recommended *not* to call in this listener any *Connext Cert* APIs that write data.

### 7.7.7 NETIO Datagram Transport

This section describes the built-in *Connext Cert* Datagram transport and how to configure it.

The built-in Datagram transport (DGRAM) is a generic transport plugin service. DGRAM is part of the *Connext Cert* core library that is compiled for a specific CPU architecture with a specific compiler. However, the DGRAM transport does not include integration with any particular network stack. Instead, the DGRAM transport provides a simplified interface which can integrate with a variety of different networking technologies.

The DGRAM plugin supports transmission and reception of RTPS messages over a connectionless network link. Note that while the DGRAM transport itself has no knowledge of the underlying network stack, the DGRAM API does not include an API related to establishing connections, such as TCP.

### Registering a Datagram interface

DGRAM is a *Connext Cert* component that can be registered with NETIO_DGRAM_InterfaceFactory_register() as shown below:

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

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

The factory gets the registry. The registry then registers the Datagram.

When a component is registered, the registration parses the DGRAM interface as the third parameter and the properties as the fourth parameter. In general, the caller is responsible to manage the memory for the properties and ensure they are valid as long as the DGRAM transport is registered. There is no guarantee that a component makes a copy.

The DGRAM interface is a component that interfaces with the *Connext Cert* core library. You must implement the `NETIO_DGRAM_InterfaceI`, which integrates with a specific network technology. This `struct` must be compliant with the `NETIO_DGRAM_InterfaceI` structure, as shown below:

```
 /* Create the DGRAM User Interface property struct */
 struct MyDgramInterfaceProperty
 {
     RTI_INT32 a_property;
     struct UTEST_Context *setting;
 } MyDgramInterfaceProperty = {10,NULL};

/* Example operation */
 struct NETIO_Interface*
 MyDgramInterface_create_instance(NETIO_Interface_T *upstream,void *property)
 {
     /* Perform operations */
 ...
    return myInterface;
 }
 ...

 /* Create the DGRAM Interface struct where each member points to it's
  * respective operation */
 RTI_PRIVATE struct NETIO_DGRAM_InterfaceI MyDgramInterface =
 {
     MyDgramInterface_create_instance,
     MyDgramInterface_get_interface_list,
     MyDgramInterface_release_address,
     MyDgramInterface_resolve_address_udpv4,
     MyDgramInterface_send,
```

(continues on next page)

(continued from previous page)

```
    MyDgramInterface_get_route_table,
    MyDgramInterface_bind_address
};
```

The following code snippet shows how to register the DGRAM interface with new parameters. The Datagram needs to register the DGRAM interface with a property that has the interface to call:

```
/* Register the transport again, using the builtin name
    */
if (!NETIO_DGRAM_InterfaceFactory_register(registry,
                "name",
                &MyDgramInterface,
                &MyDgramInterfaceProperty))
{
    /* ERROR */
}
```

**Note:** The Datagram 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 will be logged.

### Addressing a Datagram transport

The interface may also set the enabled transports to receive data, as shown below:

```
struct DDS_DomainParticipantQos dp_qos =
                                    DDS_DomainParticipantQos_INITIALIZER;

/* Datagram enable transport xyz. A second transport can be added
 * by setting the enabled_transports value to 2 and adding a second
 * transport name. enabled_transport indicates what addresses the entity is
 * listening on.
 */
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("xyz");

/* Receive discovery traffic on xyz */
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("xyz://");

/* Receive user-data traffic on xyz. */
DDS_StringSeq_set_maximum(&dp_qos.user_traffic.enabled_transports,1);
DDS_StringSeq_set_length(&dp_qos.user_traffic.enabled_transports,1);
```

(continues on next page)

(continued from previous page)

```
*DDS_StringSeq_get_reference(&dp_qos.user_traffic.enabled_transports,0) =
                                           DDS_String_dup("xyz://");
```

An address may set up peers to send messages over this interface. For example, interface xyz may set its initial peers as:

```
 /* Send discovery data on address 0x0A00020F*/
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("0x0A00020F");
```

### Setting up the Datagram UDP

The built-in Datagram transport can support UDP integration.

The built-in Datagram transport for UDP registers differently than the generic Datagram component. Use `UDP_Interface_register()` with UDP properties to create the datagram instance for UDP, as shown below:

```
struct UDP_InterfaceFactoryProperty udp_property =
                                UDP_InterfaceFactoryProperty_INITIALIZER;

/* To enable sharing this property must be set to RTI_FALSE */
udp_property->enable_interface_bind = RTI_TRUE;

/*  */
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(intf);

/* To enable multicast operations the multicast flag and multicast_interface
 * property must be set */
if (is_multicast)
{
    flags |= UDP_INTERFACE_INTERFACE_MULTICAST_FLAG;
    udp_property->multicast_interface = DDS_String_dup(intf);
}
else
{
    /* Set the mutlicast interface to NULL when not used*/
    udp_property->multicast_interface = NULL;
}

/* Add an available interface for UDP */
if (!UDP_InterfaceTable_add_entry(&udp_property->if_table,
                    address, netmask, intf_name, flags))

{
```

(continues on next page)

(continued from previous page)

```
    /* error */
}

/* Buffer properties */
udp_property->max_send_buffer_size = MAX_SEND_BUFFER_SIZE;
udp_property->max_receive_buffer_size = MAX_RECV_BUFFER_SIZE;
udp_property->max_message_size = MAX_RECV_BUFFER_SIZE;

/* Register the datagram */
if(!UDP_Interface_register(registry, "_udp", udp_property))
{
    /* error */
}
```

Enabled transports can be configured with `"_udp://"`. This will use all interfaces. Enabling a UDP is similar to generic addressing, as shown below:

```
struct DDS_DomainParticipantQos dp_qos =
                                    DDS_DomainParticipantQos_INITIALIZER;

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

/* This only requires the transport name */
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports,0) =
                                            DDS_String_dup("_udp");

/* _udp:// indicates to use all available locators */
*DDS_StringSeq_get_reference(&dp_qos.discovery.enabled_transports,0) =
                                        DDS_String_dup("_udp://");

/* _udp://10.10.0.1 would indicate to use only that address */
*DDS_StringSeq_get_reference(&dp_qos.user_traffic.enabled_transports,0) =
                                            DDS_String_dup("_udp://");
```

### Datagram shared port flag

*Connext Cert* can work with two different types of transports: ones that use shared ports and ones that do not. *Connext Cert* supports a flag for the Datagram transport, NETIO_DGRAM_INTERFACE_SHARED_PORT_FLAG, to enable the use of shared ports on a per-transport basis.

When the shared port flag is enabled, it doesn't matter which network interface the transport receives messages on; only the port matters. For example, consider a computer with two network interfaces, **A** and **B**, which is listening for messages on port **P**. As long as the message is received on any network interface capable of receiving on port **P**, the transport will accept the message; it doesn't matter if the message is received on **A** or **B**.

However, when the shared port flag is *not* enabled, it does matter which network interface the transport receives messages on. For example, consider a computer with two network interfaces, **A** and **B**, which is listening for messages on port **P** like before. However, this computer has specified that network interface **A** should only receive on port **P**. The transport will then ignore messages received on network interface **B** and port **P**.

When the transport accepts a message, the transport routes it to all relevant *DataReaders* and *DataWriters*; thus, the NETIO_DGRAM_INTERFACE_SHARED_PORT_FLAG cannot be used to control whether some DDS topics should only be accepted when received on a specific network interface. However, this feature allows different *DomainParticipants* to use the same port with different network interfaces.

### User interface

NETIO_DGRAM_InterfaceFactory_register() registers a user interface structure that is passed in via `user_intf`. The *DomainParticipant* utilizes these functions for network operations, such as creating a Datagram interface instance and getting the interface list.

Table 7.1: Structure for the User Interface

| Interface Attribute | Description |
|---|---|
| `create_instance` | Creates an instance of the NETIO_DGRAM interface. |
| `delete_instance` | Deletes an instance of the NETIO_DGRAM interface. |
| `get_interface_list` | Reads the available interfaces from the NETIO_DGRAM interface. |
| `release_address` | Instructs the NETIO_DGRAM interface to stop listening for messages on the source address. |
| `resolve_address` | Instructs the NETIO_DGRAM interface to determine if the address string is valid. |
| `send` | Instructs the NETIO_DGRAM interface to send a message. |
| `get_route_table` | Instructs the NETIO_DGRAM interface `netio_intf` to return a sequence of address and netmask pairs that this interface can send to. |
| `bind_address` | Instructs the NETIO_DGRAM interface to listen for messages on the source address. |

## 7.7.8 Zero Copy v2 Transport

The Zero Copy v2 transport enables *RTI Connext Cert* to transmit data samples without copying them internally. For an overview of this feature, see *Zero Copy Transfer*.

This section outlines the basic steps required to enable the Zero Copy v2 transport in an application. All the example code shown below is taken from the Zero Copy HelloWorld example on GitHub.

### Generate type support files

First, identify types that require Zero Copy transfer and annotate them with the `@transfer_mode(SHMEM_REF)` annotation. See the HelloWorld.idl file from the Zero Copy HelloWorld example:

```
@transfer_mode(SHMEM_REF)
struct HelloWorld {
    @key long  id;
    long  data[100];
};
```

*rtiddsgen* generates additional TypePlugin code when a type is annotated with `@transfer_mode(SHMEM_REF)` in the IDL files. This code allows a *DataWriter* and a *DataReader* to communicate using a reference to the sample in shared memory.

**Note:** Zero Copy v2 for *Connext Cert* only supports contiguous data types; this means that fixed-size arrays are supported, but sequences and strings are not.

Next, generate type support files with the following command:

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

The generated files will appear in the same directory as the type file.

### Initialize the Zero Copy v2 transport

Before the Zero Copy v2 transport can be used, it must be initialized. This must be done before creating a *DomainParticipant* and after registering the *DataReader* and *DataWriter* history plugins. The order of these operations is important because the Zero Copy v2 transport will perform actions on the history components during initialization.

The following example code from **HelloWorldApplication.c** demonstrates how to initialize the Zero Copy v2 transport:

```
if (!NDDS_Transport_ZeroCopy_initialize(registry, NULL, NULL))
{
    printf("failed to initialize zero copy\n");
    /* handle error */
}
```

### Register the Zero Copy v2 transport

The Zero Copy v2 transport needs a notification mechanism to send notifications from *DataWriters* to *DataReaders*. You can use the reference notification mechanism provided by RTI, or implement your own.

Once the Zero Copy transport is initialized, configure the `notif` interface factory with the ZCOPY_NotifInterfaceFactoryProperty property. This property has three fields you need to set:

- max_samples_per_notif: The number of samples processesed per notification. By default this value is 1. Note that a high value may starve other threads from progressing.
- user_intf: This is the implementation of your chosen notification mechanism. It is populated automatically if you are using the reference implementation.
- user_property: Any properties associated with your chosen notification mechanism. *Connext Cert* treats this as an opaque pointer.

**Note:** For reference, RTI provides a posix implementation of the notification mechanism. This mechanism is based on a monitor implemented in shared memory. This documentation assumes that you are using the reference implementation.

When using the notification mechanism provided by RTI, the user_property mentioned above is resolved to `ZCOPY_NotifMechanismProperty`. Both of these properties are required to configure the transport.

See the following example from **HelloWorldApplication.c**:

```
struct ZCOPY_NotifInterfaceFactoryProperty  notif_prop;
struct ZCOPY_NotifMechanismProperty notif_mech_prop;
notif_mech_prop.intf_addr = 0;
notif_prop.user_property = &notif_mech_prop;
notif_prop.max_samples_per_notif = 1;
```

For more information on these properties, see the *Configuration* section.

Finally, call `ZCOPY_NotifMechanism_register()` (a utility function on the reference notification mechanism) to register the Zero Copy v2 transport with the default notification mechanism. This makes it available for use. The following example registers a `notif` with the name `NETIO_DEFAULT_NOTIF_NAME`:

```
if (!ZCOPY_NotifMechanism_register(registry, NETIO_DEFAULT_NOTIF_NAME, &notif_prop))
{
    printf("failed to register notif\n");
    goto done;
}
```

### Enable transports

With the specific notification mechanism registered, you can enable the Zero Copy and UDP transports for the *DomainParticipant*. Consider the following example code:

```
if (!DDS_StringSeq_set_maximum(&dp_qos.transports.enabled_transports, 2))
{
    printf("ERROR: Failed to set transports.enabled_transports maximum\n");
    goto done;
}
if (!DDS_StringSeq_set_length(&dp_qos.transports.enabled_transports, 2))
{
    printf("ERROR: Failed to set transports.enabled_transports length\n");
    goto done;
}
/* UDP and Notification are enabled*/
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports, 0) =
        DDS_String_dup("notif");
*DDS_StringSeq_get_reference(&dp_qos.transports.enabled_transports, 1) =
        DDS_String_dup(MY_UDP_NAME);

/* Discovery takes place over UDP */
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("_udp://");

/* User data uses Notification Transport */
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("notif://");
```

**Note:** The UDP transport must be registered while using Zero Copy v2 transfer because it is used during DDS Discovery (see *Discovery* for more details).

### Manage DDS samples

When using the Zero Copy v2 transport, each *DataWriter* manages a pool of samples, and the application must obtain samples from this pool using get_loan(). We can see this in the following example:

```
hw_datawriter = HelloWorldDataWriter_narrow(datawriter);
retcode = HelloWorldDataWriter_get_loan(hw_datawriter, &sample);
if (retcode != DDS_RETCODE_OK)
{
    printf("ERROR: Failed to loan sample\n");
}
retcode = HelloWorldDataWriter_write(hw_datawriter, sample, &DDS_HANDLE_NIL);
```

(continues on next page)

(continued from previous page)

```
if (retcode != DDS_RETCODE_OK)
{
    printf("ERROR: Failed to write to sample\n");
}
```

As seen above, the *DataWriter* **must** get a loan before each write call; it cannot write a loaned sample multiple times. The *DataWriter* does not need to explicitly return any loan to the pool, since this is managed by the middleware. However, if a loaned sample will not be written, it can be discarded with discard_loan()..

**Warning:** It is not possible to write a sample that has not been obtained with get_loan().

**Note:** A Zero Copy-enabled *DataWriter* can also send samples using other transports (such as UDPv4) to non-Zero Copy *DataReaders*. When a *DataWriter* uses both the Zero Copy v2 transport and a transport which uses serialized data (such as UDP), the same sample is sent over all transports.

This may adversely affect performance, since the sample must be serialized for network transmission even if it is in shared memory. For best performance, you should consider an architecture where a *DataWriter* matches either with Zero Copy-enabled or non Zero Copy-enabled *DataReaders*, but not both.

On the *DataReader* side, Zero Copy v2 application code is identical to subscribing applications not using Zero Copy.

When a *DataReader* calls read() or take() and receives samples, it is being given samples that are loaned from the *DataWriter*'s pool. Thus, failing to return the loan when the sample is no longer needed will deplete the available samples in the pool, eventually causing calls to get_loan() to fail.

### Configuration

*Connext Cert* Zero Copy v2 introduces some properties unique to its functionality. The following properties are always required:

- max_samples_per_notif
- user_intf[1]
- user_property[2]

The following properties are required if you are using the reference implementation of the notification mechanism for Zero Copy v2. These are essentially a default set of user-defined properties;

[1] This property is only required if you choose to implement your own notification mechanism and not use the reference implementation provided by RTI.

[2] Resolves to `ZCOPY_NotifMechanismProperty` when using the reference notification mechanism.

if you are using your own notification mechanism, you can set your own user-defined properties as needed.

Table 7.2: Default User-Defined Properties for Zero Copy v2

| Property Name | Description | Default |
|---|---|---|
| `user_property.intf_addr` | Sets the interface address for this instance. Should be unique for all instances in the system. | 0 |
| `user_property.thread_prop.stack_size` | Sets the stack size for the receive thread created for receiving notifications. | `OSAPI_THREAD_PROPERTY_DEFAULT` |
| `user_property.thread_prop.priority` | Sets the priority for the receive thread created for receiving notifications. | `OSAPI_THREAD_PROPERTY_DEFAULT` |
| `user_property.thread_prop.options` | Sets any thread options supported by the thread API implementation. | `OSAPI_THREAD_PROPERTY_DEFAULT` |
| `user_property.max_receive_ports` | Sets the maximum number of receive ports that can be opened. Used as part of the `user_intf.bind` function. | 2 |
| `user_property.max_routes` | Sets the maximum number of outgoing routes. Used as part of the `user_intf.add_route` function. | 32 |

The following additional properties are only required if you are using your own notification mechanism for Zero Copy v2, NOT the reference implementation. Refer to the Safety Integration Manual for more information on these properties.

Table 7.3: User Interface Properties for Zero Copy v2

| Property Name | Type | Description |
|---|---|---|
| `user_intf. create_instance` | NETIO_Interface_T | Creates an instance of the notification mechanism interface with upstream as the owner. |
| `user_intf. delete_instance` | NETIO_Interface_T | Deletes an instance of the notification mechanism. |
| `user_intf. get_route_table` | RTI_BOOL | Instructs the notification mechanism interface `netio_intf` to return a sequence of address and netmask pairs this interface can send to. |
| `user_intf. reserve_address` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to setup resources for listening to messages on the address `src_addr` and return a `port_entry_out`. The `port_entry_out` will be provided to a `bind` call. |
| `user_intf. release_address` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to release resources for listening to messages on the address `src_addr`. |
| `user_intf. resolve_address` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to determine if the address string `address_string` is a valid address and return the result in `address_value`. |
| `user_intf. add_route` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to add a route from the source to the destination. The `route_entry_out` will be provided to you later when calling `send`. |
| `user_intf. delete_route` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to remove a route from the source to the destination. |
| `user_intf.bind` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to start listening for messages on the address specified by `src_addr` on the given `port_entry`. |
| `user_intf. unbind` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to stop listening for messages on the address specified by `src_addr`. |
| `user_intf.send` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to send a notification to the destination using the `route_entry`. |
| `user_intf. notify_recv_port` | RTI_BOOL | Instructs the notification mechanism interface specified by `user_intf` to notify the receive port specified by `port_entry`. |

#### Using multiple Zero Copy v2 transport instances

The platform-independent Zero Copy v2 transport supports multiple instances, provided that the user-defined, platform-specific implementation of the `notif` interface implements a way to track different interface addresses. In this case, each Zero Copy v2 transport instance should be registered with uniquely different names and properties.

When multiple instances of the Zero Copy v2 transport exist, individual *DataReaders* and *DataWriters* can be configured to use a specific instance of the Zero Copy v2 transport. This configuration is done in the entity's enabled_transports QoS configuration. For more information, see *Transport Registration*.

## 7.8 Discovery

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

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

### 7.8.1 What is Discovery?

Discovery is the behind-the-scenes way in which *RTI Connext Cert* 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 Cert* 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 Cert* on the sending and receiving sides.

Note that this chapter is shared between *Connext Micro* and *Connext Cert*. However *Connext Cert* only supports static endpoint discovery described in *Static Discovery Plugin*.

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

#### Simple Participant Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Participant Discovery Protocol (SPDP) and is common for both dynamic and static endpoint discovery.

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, *Connext Cert* 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. 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. The entityID is set by *Connext Cert* (you may be able to change it in a future version).

### 7.8.2 Configuring Participant Discovery Peers

An *RTI Connext Cert 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.

### The Peer Address

A peer descriptor string of the initial_peers sequence defines the interface and address of the locator to which to send, as well as the indices of participants to which to send. The peer descriptor format is:

```
< > denotes optional
[ ] denotes range or discreet values, unless enclosed in ''
    which means a literal.

ADDRESS = <PREFIX://><ADDRESS> |
          @<PREFIX://><ADDRESS> |
          INDEX@<PREFIX>://<ADDRESS>

INDEX = INTEGER | '[' INTEGER ']' | '[' INTEGER-INTEGER ']' | '[' -INTEGER ']'

PREFIX = [a-zA-Z_][0-9a-zA-Z_]+

INTEGER = DEC_INTEGER | HEX_INTEGER

DEC_INTEGER = [0-9]+

HEX_INTEGER = [0x|0X][0-9a-fA-F]+

ADDRESS = 0 or more 8bit characters
```

Note that peer descriptors use zero-based indexing; the first peer index is 0, not 1. Also note that while the `PREFIX` is marked optional, it should always be used.

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 ports 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 one or more ports 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.

For example:

```
DDS_StringSeq_set_maximum(&dp_qos.discovery.initial_peers, 5);
DDS_StringSeq_set_length(&dp_qos.discovery.initial_peers, 5);

/* If the index is not specified, it defaults to 5, thus sending to
 * the first 6 participant IDs on at IP address 192.168.1.1 using
 * the transport registered as _udp.
 */
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 0) =
     DDS_String_dup("_udp://192.168.1.1");

/* Only send participant annoucements to multicast address 239.255.0.1
 * using the transport registered as _udp. Note that for multicast
```

(continues on next page)

(continued from previous page)

```
 * addresses the index is not relevant since it is a shared address.
 */
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 1) =
     DDS_String_dup("_udp://239.255.0.1");

/* Send annoucements to participant ID 1,2,3, and 4 on 10.10.30.101
 * using the transport registered as _udp.
 */
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 2) =
     DDS_String_dup("[0-4]@_udp://10.10.30.101");

/* Send annoucements to participant ID 2 on address 10.10.30.102
 * using the transport registered as _udp.
 */
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 3) =
     DDS_String_dup("[2]@_udp://10.10.30.102");

/* Send annoucements to participant ID 0-8 on address 10.10.30.102
 * using the transport registered as _udp.
 */
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 4) =
     DDS_String_dup("8@_udp://10.10.30.102");
```

### 7.8.3 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.

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.

### 7.8.4 Configuring Discovery Data Reception

In order to *receive* discovery and user data, it is necessary to configure the `DomainParticipantQos.discovery.enabled_transports` sequence. This is a sequence of transport addresses to listen for discovery data on, and is sent as part of the participant annoucements. Other *DomainParticipants* will send to these addresses.

The address format for configuring data reception uses the following format:

```
< > denotes optional
[ ] denotes range or discreet values, unless enclosed in ''
    which means a literal.

```

(continues on next page)

(continued from previous page)

```
ADDRESS = <PREFIX://><ADDRESS>

PREFIX = [a-zA-Z_][0-9a-zA-Z_]+

ADDRESS = 0 or more 8bit characters
```

Note that while the `PREFIX` is marked optional, it should always be used.

For example, to receive on a single unicast address:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.discovery.enabled_transports, 1);
DDS_StringSeq_set_length(&DomainParticipantQos.discovery.enabled_transports, 1);

/* Receive on the unicast address 192.168.1.1 using the transport registered
 * as _udp.
 */
*DDS_StringSeq_get_reference(&DomainParticipantQos.discovery.enabled_transports, 0) =
                              DDS_String_dup("_udp://192.168.1.1");
```

To receive on all unicast addresses allowed by the transport:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.discovery.enabled_transports, 1);
DDS_StringSeq_set_length(&DomainParticipantQos.discovery.enabled_transports, 1);

/* Receive on all unicast addresses allowed by the transport registered
 * as _udp. This is not recommended if more than 4 network interfaces are
 * allowed as it is non-deterministic which interfaces will be used.
 */
*DDS_StringSeq_get_reference(&DomainParticipantQos.discovery.enabled_transports, 0) =
                              DDS_String_dup("_udp://");
```

To receive on one unicast address and one multicast address:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.discovery.enabled_transports, 2);
DDS_StringSeq_set_length(&DomainParticipantQos.discovery.enabled_transports, 2);

*DDS_StringSeq_get_reference(&DomainParticipantQos.discovery.enabled_transports, 0) =
                              DDS_String_dup("_udp://192.168.1.1");

*DDS_StringSeq_get_reference(&DomainParticipantQos.discovery.enabled_transports, 1) =
                              DDS_String_dup("_udp://239.255.0.1");
```

To receive on one multicast address:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.discovery.enabled_transports, 1);
DDS_StringSeq_set_length(&DomainParticipantQos.discovery.enabled_transports, 1);

*DDS_StringSeq_get_reference(&DomainParticipantQos.discovery.enabled_transports, 0) =
                              DDS_String_dup("_udp://239.255.0.1");
```

### 7.8.5 Configuring User Data Reception

In order to *receive* discovery and user data, it is necessary to configure the `DomainParticipantQos.user_traffic.enabled_transports` sequence. This is a sequence of default transport addresses to listen for user data on (unless a *DataReader* or *DataWriter* specifies its own address; see *Configuring Enabled Transports per DataReader or DataWriter* below) and is sent as part of the participant annoucements. Other *DomainParticipants* will send to these addresses.

The address format for configuring data reception uses the following format:

```
< > denotes optional
[ ] denotes range or discreet values, unless enclosed in ''
    which means a literal.

ADDRESS = <PREFIX://><ADDRESS>

PREFIX = [a-zA-Z_][0-9a-zA-Z_]+

ADDRESS = 0 or more 8bit characters
```

Note that while the `PREFIX` is marked optional, it should always be used.

For example, to receive on a single unicast address:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.user_traffic.enabled_transports, 1);
DDS_StringSeq_set_length(&DomainParticipantQos.user_traffic.enabled_transports, 1);

/* Receive on the unicast address 192.168.1.1 using the transport registered
 * as _udp.
 */
*DDS_StringSeq_get_reference(&DomainParticipantQos.user_traffic.enabled_transports, 0) =
                             DDS_String_dup("_udp://192.168.1.1");
```

To receive on all unicast addresses allowed by the transport:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.user_traffic.enabled_transports, 1);
DDS_StringSeq_set_length(&DomainParticipantQos.user_traffic.enabled_transports, 1);

/* Receive on all unicast addresses allowed by the transport registered
 * as _udp. This is not recommended if more than 4 network interfaces are
 * allowed as it is non-deterministic which interfaces will be used.
 */
*DDS_StringSeq_get_reference(&DomainParticipantQos.user_traffic.enabled_transports, 0) =
                             DDS_String_dup("_udp://");
```

To receive on one unicast address and one multicast address:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.user_traffic.enabled_transports, 2);
DDS_StringSeq_set_length(&DomainParticipantQos.user_traffic.enabled_transports, 2);

*DDS_StringSeq_get_reference(&DomainParticipantQos.user_traffic.enabled_transports, 0) =
                             DDS_String_dup("_udp://192.168.1.1");
```

(continues on next page)

(continued from previous page)

```

*DDS_StringSeq_get_reference(&DomainParticipantQos.user_traffic.enabled_transports, 1) =
                             DDS_String_dup("_udp://239.255.0.1");
```

**Note:** When both multicast and unicast is specified, the following rules are used:

- New data is sent over multicast.
- Retransmissions are sent over unicast.

To receive on one multicast address:

```
DDS_StringSeq_set_maximum(&DomainParticipantQos.user_traffic.enabled_transports, 1);
DDS_StringSeq_set_length(&DomainParticipantQos.user_traffic.enabled_transports, 1);

*DDS_StringSeq_get_reference(&DomainParticipantQos.user_traffic.enabled_transports, 0) =
                             DDS_String_dup("_udp://239.255.0.1");
```

### 7.8.6 Configuring Enabled Transports per DataReader or DataWriter

A *DataReader* or *DataWriter* can specify its own addresses in the `DataReaderQos.transport.enabled_transports` and `DataWriterQos.transport.enabled_transports` policies. The address format is exactly the same as for `DomainParticipantQos.user_traffic.enabled_transports`, with the restriction that a *DataWriter* can only specify its own unicast addresses.

### 7.8.7 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 Cert* provides two ways of determinig endpoint information of other *DomainParticipants*: *Dynamic Discovery Plugin* and *Static Discovery Plugin*.

#### Dynamic Discovery Plugin

**Note:** The Dynamic Discovery Plugin is not available in *Connext Cert*. The description is only included for completeness and comparison with the *Static 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.

**Note:** *RTI Connext* uses the acronyms SPDP and SEDP to distinguish between the two phases of Simple Discovery: participant and endpoint phases (see Discovery in the Core Libraries User's Manual). *RTI Connext Cert* uses the acronyms DPSE and DPDE to distinguish between the static and dynamic endpoint discovery plugins available in *RTI Connext Cert*. The DPSE plugin implements the SPDP protocol and DPDE implements the SPDP and SEDP protocol.

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

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*.

**Note:** *RTI Connext* uses the acronyms SPDP and SEDP to distinguish between the two phases of Simple Discovery: participant and endpoint phases (see Discovery in the Core Libraries User's Manual). *RTI Connext Cert* uses the acronyms DPSE and DPDE to distinguish between the static and dynamic endpoint discovery plugins available in *RTI Connext Cert*. The DPSE plugin implements the SPDP protocol and DPDE implements the SPDP and SEDP protocol.

Please refer to the C API Reference 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));
```

(continues on next page)

(continued from previous page)

```
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*.

### 7.8.8 Asymmetric Matching and Lost Samples

The DDS discovery process is necessary to establish communication between a *DataWriter* and a *DataReader*. However, it is important to understand that DDS applications do not connect to each other; there is no handshake protocol to ensure that a *DataReader* is ready to receive data from a *DataWriter*. Thus, it is possible that a *DataWriter* matches a *DataReader* before the *DataReader* matches the *DataWriter* (and vice versa). For this reason, it is possible that data published by a *DataWriter* is not received by the *DataReader*, even on a local network.

The reason for this asymmetric behavior can be for any number of reasons, such as, but not limited to:

- Network delays
- Packets taking different paths through the network
- Address resolution delays
- OS scheduling

DDS offers some solutions to mitigate this problem, e.g., the DURABILITY QoS policy, but in other cases it may be necessary for applications to implement their own synchronization protocols.

## 7.9 Generating Type Support with rtiddsgen

### 7.9.1 Why Use rtiddsgen?

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

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

### 7.9.2 IDL Type Definition

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

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

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

### 7.9.3 Generating Type Support

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

- `RTIMEHOME` sets the path of the *Connext Cert* installation directory: `NDDSHOME/rti_connext_dds_cert-2.4.15`
- `RTIMEARCH` sets the platform architecture (e.g. i86Linux2.6gcc4.4.5 or i86Win32VS2010)

Note that a JRE is shipped with *Connext Cert* 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):

```
.. only:: not cert
```

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

> $rti_connext_micro_root/rti_me_cert.1.0/rtiddsgen/scripts/rtiddsgen -micro -language C -replace UserType.idl

#### Notes on Command-Line Options

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

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

```
> rtiddsgen -micro -help
```

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

#### Generated Type Support Files

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

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

Please note that when compiling for *Connext Cert*, -DRTI_CERT must be passed to the compiler.

### 7.9.4 Using custom data-types in Connext Cert Applications

A *Connext Cert* 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"
#include "HelloWorldSupport.h"

/* ... */

retcode = HelloWorldTypeSupport_register_type(application->participant,
                                              "HelloWorld");
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.

### 7.9.5 Customizing generated code

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

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

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

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

### 7.9.6 Unsupported Features of rtiddsgen with Connext Cert

*Connext Cert* 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 Cert*.

## 7.10 Thread Safety

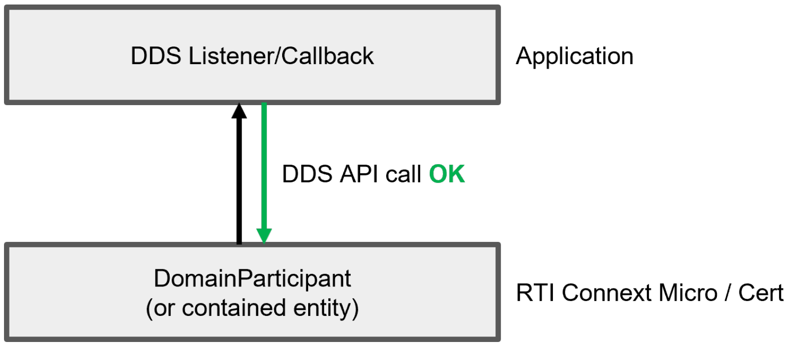
All public APIs have a note about thread-safety included in the API reference manuals. It is important that an application does not violate thread-safety guidelines. *RTI Connext Cert* creates one critical section per *DomainParticipant* to protect resources used by the *DomainParticipant*.

**Note:** Although *Connext Cert* 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.

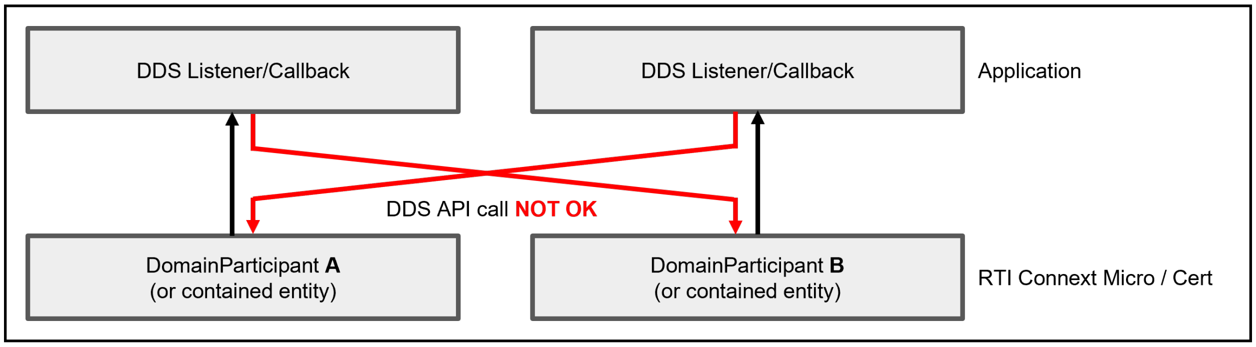
### 7.10.1 Calling DDS APIs from listeners and callbacks

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 a listener calls *Connext Cert* DDS APIs from the **same** *DomainParticipant* (and contained entities) that the listener was called from, as shown in the diagram below:



**Warning:** It is **not** safe to call DDS APIs from a **different** *DomainParticipant* than the one listener was called from, as shown in the diagram below. This may result in a deadlock situation.



**Warning:** 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.

### 7.10.2 Calling DDS APIs from a type-plugin

A user type-plugin that is registered with the *DomainParticipant* is subject to the following rules:

- The key kind is constant.
- The plugin is constant for a given DDS entity (*Topic*, *DataWriter*, or *DataReader*).
- The plugin data must be protected if thread safety is a concern, as it is user data and not protected by *Connext Cert*.

**Note:** A type-plugin generated from an IDL file with the *rtiddsgen* IDL compiler included with *Connext Cert* will satisfy these rules.

## 7.11 Memory Model

### 7.11.1 Introduction

*Connext Cert* is designed for use in real-time systems and uses a predictable and deterministic memory manager to ensure that memory growth is not unbounded, OS memory fragmentation is eliminated and memory usage can be determined a-priori. The advantage with this design is that proper operation is ensured as soon as steady state has been reached. However, it also places an additional burden on the system designer to properly configure each resource-limit. The purpose of this document is to describe all resource-limits in *Connext Cert*, what the behavioral impact is, and what the impact on memory usage is.

### 7.11.2 Dynamic Memory Allocation

*Connext Cert* allocates heap memory to create internal data-structures. It is important to know that *Connext Cert* manages memory allocated from the system heap using its own internal memory management, and only returns memory allocated from the system back to the system when something is deleted. That is, if an application never deletes anything, no memory is returned to the system.

As a rule of thumb, in *Connext Cert* the only APIs that allocate heap memory are:

- DDS_DomainParticipantFactory_get_instance().
- Those that contain the word "new", i.e. DDS_WaitSet_new().
- Those that contain the word "create", i.e. DDS_DomainParticipantFactory_create_participant().

And the only APIs that free memory are:

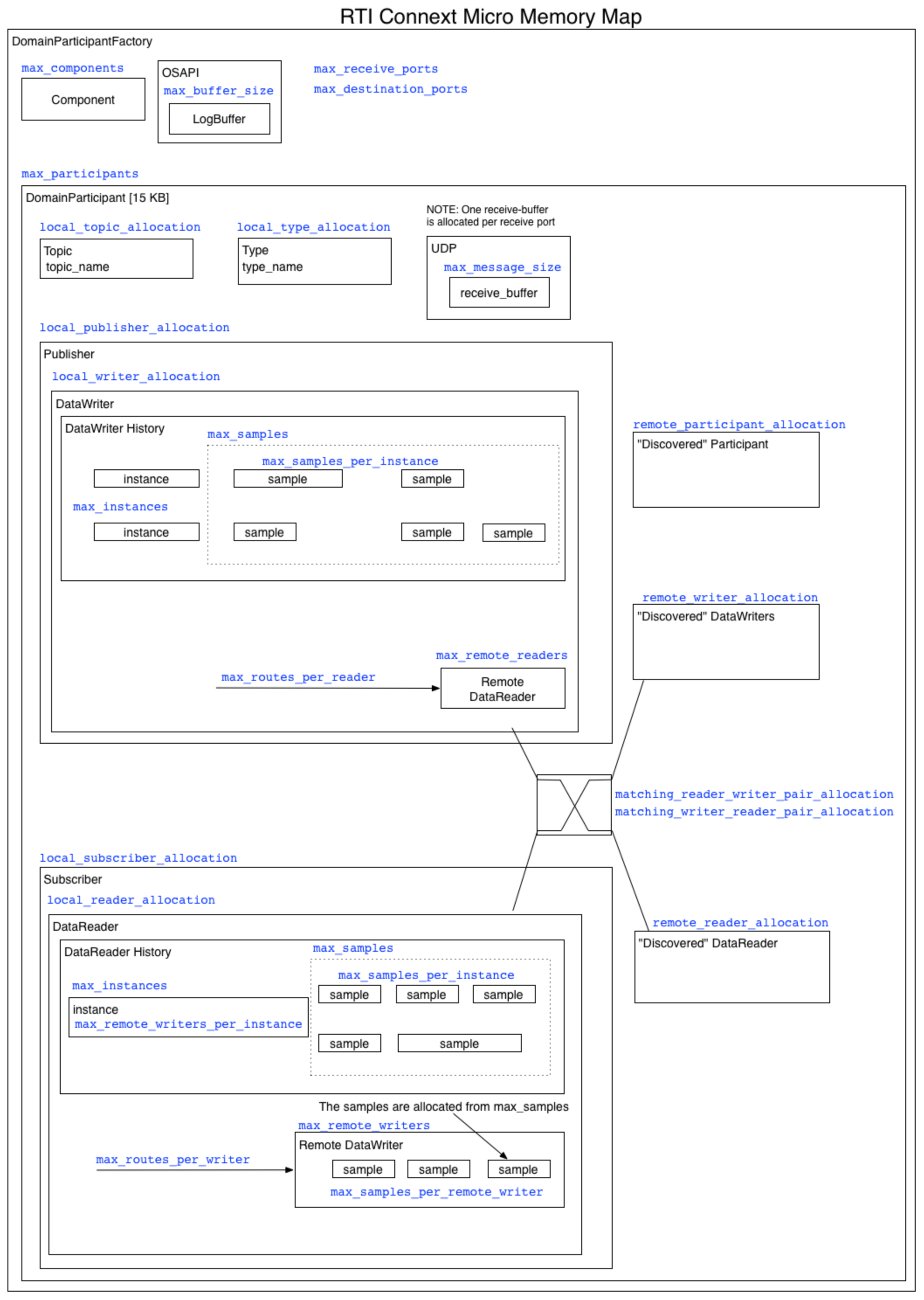
- DDS_DomainParticipantFactory_finalize_instance().
- Those that contain the word "delete", i.e. OSAPI_Waitset_delete().
- Those that contain the word "free", i.e. DDS_String_free().

The *Connext Cert* user is supposed to use API function which allocate memory only during system initialization. This avoid to run into situations where a resource can not be allocated after the initialization phase.

*Connext Cert* does not support dynamically allocating resources beyond the initial configuration. That is, all resource limits must be finite. This restriction may be removed in a future version.

### 7.11.3 Resource Limits

All resource-limits in *Connext Cert* are specified in a Qos policy or property. The figure below illustrates where the various resource-limits are applied, and the section Resource Limits describes each user-adjustable resource limit in more detail. All numbers in blue correspond to a resource limit.


RTI Connext Micro Memory Map
DomainParticipantFactory
max_components
Component
OSAPI
max_buffer_size
LogBuffer
max_receive_ports
max_destination_ports
max_participants
DomainParticipant [15 KB]
local_topic_allocation
Topic
topic_name
local_type_allocation
Type
type_name
NOTE: One receive-buffer is allocated per receive port
UDP
max_message_size
receive_buffer
local_publisher_allocation
Publisher
local_writer_allocation
DataWriter
DataWriter History
max_samples
max_samples_per_instance
instance
sample
max_instances
max_remote_readers
max_routes_per_reader
Remote DataReader
remote_participant_allocation
"Discovered" Participant
remote_writer_allocation
"Discovered" DataWriters
matching_reader_writer_pair_allocation
matching_writer_reader_pair_allocation
local_subscriber_allocation
Subscriber
local_reader_allocation
DataReader
DataReader History
max_samples
max_instances
max_samples_per_instance
instance
max_remote_writers_per_instance
sample
The samples are allocated from max_samples
max_remote_writers
Remote DataWriter
max_routes_per_writer
max_samples_per_remote_writer
remote_reader_allocation
"Discovered" DataReader

## 7.12 Zero Copy Transfer

Zero Copy transfer enables *RTI Connext Cert* to transmit data samples without copying them internally, similar to Zero Copy Transfer Over Shared Memory in *Connext Professional*. This offers several benefits, including higher throughput of user data, reduced latency between DDS endpoints (compared to other transports that send serialized data, such as UDP), and decoupling sample size from latency. This is particularly useful in applications with large sample sizes, such as image or lidar point cloud data.

At a high level, Zero Copy transfer works by a *DataWriter* and *DataReader* accessing the same shared memory; see Figure 7.4 below. A Zero Copy-enabled *DataWriter* creates a structure in a shared memory region and allocates samples from its pool to shared memory. When samples are published by the *DataWriter*, matching *DataReaders* are notified via a transport that new samples are available in shared memory. The *DataReader* then accesses the samples in shared memory using the standard DDS read/take APIs. Note that the *DataReader* and *DataWriter* must be co-located—that is, within the same operating system instance.

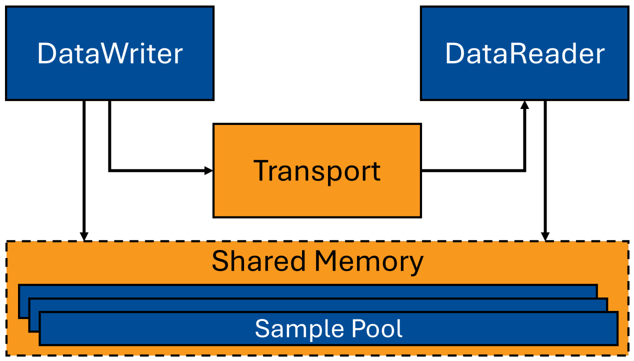


Figure 7.4: Zero Copy Transfer in *Connext Cert*

### 7.12.1 Compatibility

**Note:** In this documentation, we may refer to the Zero Copy transfer method in *Connext Cert* as "Zero Copy v2". This is because *Connext Cert* contains a subset of features from *Connext Micro*, and *Connext Micro* has two different methods for performing Zero Copy transfer: v1 and v2. *Connext Cert* only supports Zero Copy v2.

Zero Copy v2 is compatible with *Connext Micro* applications IF they are also using the Zero Copy v2 method. However, Zero Copy v2 is NOT compatible with the Zero Copy v1 method in *Connext Micro* (sometimes referred to as "Zero Copy transfer over Shared Memory" in documentation) or the Shared Memory Transport (SHMEM). If you are unsure if your *Connext Micro* version supports Zero Copy v2, refer to the documentation in your *Connext Micro* installation.

Zero Copy v2 is NOT interoperable with the Zero Copy Transfer Over Shared Memory feature in *Connext Professional*.

### 7.12.2 Overview

Zero Copy 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 you. Because the two processes share different memory spaces, A* and B* will be different but they will point to the same place in RAM.

This feature requires using new RTI DDS Extension APIs:

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

### 7.12.3 Getting started

To enable Zero Copy transfer, 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 a type is annotated with the PLAIN language binding (e.g., `@language_binding(PLAIN)` in IDL).

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

2. *Register the Zero Copy v2 transport*. References will be sent across this 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
```

### Writing samples

The following code illustrates how to write 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);
}
```

### Reading samples

The following code illustrates how to read samples annotated with `@transfer_mode(SHMEM_REF)`:

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

/* process sample here */
/* NEW API
   IMPORTANT - is_data_consistent will always return true when ZC v2 is being used
 */

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

if (dds_rc == DDS_RETCODE_OK)
{
     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.
```

(continues on next page)

(continued from previous page)

```
        */
    }
}
```

### 7.12.4 Synchronizing samples

Zero Copy v2 provides synchronization between the *DataWriter* and *DataReader*. Since the queue size is limited, samples are reused once the queue is full. However, if a *DataWriter* modifies a sample before the *DataReader* has accessed it, that sample will not be presented to you. Additionally, samples currently being read by the *DataReader* are locked, preventing the *DataWriter* from accessing them.

Consider the following example with `max_samples = 1` (internally, the middleware will allocate 2 samples):

```
ddsrc = FooDataWriter_get_loan(dw, &sample); /* returns pointer to sample 1 */
sample->value = 10000;
ddsrc = FooDataWriter_write(datawriter, sample, &DDS_HANDLE_NIL);

ddsrc = FooDataWriter_get_loan(dw, &sample); /* returns pointer to sample 2 */
sample->value = 20000;
ddsrc = FooDataWriter_write(datawriter, sample, &DDS_HANDLE_NIL);

/* Both samples are now available to the user, but the Reader may not have accessed them␣
↪yet. */

ddsrc = FooDataWriter_get_loan(dw, &sample); /* returns pointer to sample 1 */
sample->value = 30000;
ddsrc = FooDataWriter_write(datawriter, sample, &DDS_HANDLE_NIL);
```

If the *DataReader* takes all the samples:

```
FooDataReader_take(reader,
                  &sample_seq, &info_seq,
                  DDS_LENGTH_UNLIMITED,
                  DDS_ANY_SAMPLE_STATE,
                  DDS_ANY_VIEW_STATE,
                  DDS_ANY_INSTANCE_STATE);
```

It will only receive two samples with values of `20000` and `30000`, respectively.

If you are currently accessing both samples by calling FooDataReader_read() or FooDataReader_take(), they will be locked. Any attempt to call FooDataWriter_get_loan() on the *DataWriter* will result in an `OUT_OF_RESOURCES` error.

### 7.12.5 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 the FooDataWriter. Otherwise, subsequently calling 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.

## 7.13 Message Integrity Checking

*Connext Cert* uses the DDS-I/RTPS protocol for communication between DDS applications, and RTPS messages are sent and received by a transport. When an RTPS message is sent across a communication link, such as Ethernet, it is possible that some bits may change value. These errors may cause communication failures or incorrect data to be received. In order to *detect* these types of errors, transports such as UDP often include a checksum to validate the integrity of the data: a sender adds the checksum to the transmitted data and the receiver validates that the calculated checksum for the received data matches the checksum received from the sender. If the checksums are different, a data corruption has occurred.

By default, *Connext Cert* relies on the underlying transport, such as UDP, to handle data integrity checking. However, the underlying transport may not provide sufficient integrity checking, or may itself introduce errors that *Connext Cert* must be able to detect regardless of the transport.

In order to address both of these scenarios for *any* transport, *Connext Cert* supports RTPS message integrity checking by adding a checksum to the RTPS message itself. This chapter describes the setup and default options to access this feature.

For information on how to write custom checksum functions, please refer to *RTPS*.

### 7.13.1 RTPS Checksum

*Connext Cert* implements checksum validation on a complete RTPS message. A typical RTPS message without a checksum has the following structure:

```
+--------+------------+------------------------+------------+
| Header | Submessage | ...... submessages .... | Submessage |
+--------+------------+------------------------+------------+
```

When the message integrity checking feature is enabled, the structure of the RTPS message changes as illustrated below:

```
+--------+------------+------------+-----------------+------------+
| Header | Checksum   | Submessage | .. submessages ..| Submessage |
+--------+------------+------------+-----------------+------------+
```

The sender calculates the checksum for the entire message with a checksum field set to 0 and places the result in the checksum field.

The receiver saves the the received checksum, sets the received checksum field to 0, and calculates the checksum for the entire message. It then compares the calculated checksum with the received checksum. If the checksums differ, the entire RTPS message is considered corrupted.

Note that the checksum is used *only* for error *detection* and *not* for error *correction*.

### 7.13.2 Configurations

You can configure your application to define which algorithms to use and validate as well as the requirements enforced by the participant when communicating with other participants using the *DDS_WireProtocolQosPolicy*.

Configuring the message integrity checking consists of the two parts:

1. Selecting the checksum algorithm.
2. Configuring how a participant applies the checksums.

#### Selecting a checksum algorithm

*Connext Cert* supports three built-in algorithms and can be configured to use any of the following algorithms:

1. DDS_CHECKSUM_BUILTIN32: CRC-32 As defined by ISO/IEC 13239:2002.
2. DDS_CHECKSUM_BUILTIN64: CRC-64 As defined by ISO/IEC 13239:2002.
3. DDS_CRC_BUILTIN128: MD5 Message Digest

The CRC functions have the following properties:

| Checksum | Polynom | Initial Value | Input Reflected | Output Reflected | XOR Value |
|---|---|---|---|---|---|
| CRC-32 | 0x04c11db7 | 2^32 - 1 | true | true | 2^32 - 1 |
| CRC-64 | 0x1b | 2^64 - 1 | true | true | 2^64 -1 |

In addition, four custom algorithms can implemented and used:

1. DDS_CHECKSUM_CUSTOM32
2. DDS_CHECKSUM_CUSTOM64
3. DDS_CHECKSUM_CUSTOM128
4. DDS_CHECKSUM_CUSTOM256

Please refer to *RTPS* for information on how to implement custom checksum functions.

### Configuring the DDS DomainParticipant

The RTPS message integrity feature is configured in the *DDS_WireProtocolQosPolicy* for a participant. This QoS determines which RTPS checksum should be allowed, and if checksums should be sent and/or validated.

The following three fields determine how a participant uses RTPS checksums:

- *compute_crc* - This configures the participant to send a checksum with each RTPS message. Which checksum to send is determined by *computed_crc_kind*.
- *check_crc* - This configures the participant to verify the checksum in each received RTPS message if the checksum is present. If the checksum is valid, the message is accepted; otherwise, the message is dropped. If a message is received *without* a checksum, it is accepted and processed.
- *require_crc* - This configures the participant to *require* that a checksum is present in the receiving packet. Messages without a checksum are dropped without further processing. Note that this option is orthogonal to the *check_crc options*. This option only requires that a checksum is included, it does not validate it. To validate and only accept messages with a checksum, *both* check_crc and require_crc must be *true*.

The following two fields determine which checksums are used:

- *computed_crc_kind* - The checksum type to include in each RTPS message when *compute_crc* is *true*.
- *allowed_crc_mask* - A mask of all checksum algorithms that the participant can verify. This allows the participant to receive messages from other participants with a different *computed_crc_kind*. A participant will ignore a participant that is sending a checksum that it cannot validate.

For example, the following snippet shows how to configure the participant to:

- Send all messages (except the participant announcements; see the *Participant Discovery and Participant Compatibility* section below) with *DDS_CHECKSUM_BUILTIN64*.
- Accept *DDS_CHECKSUM_BUILTIN32*, *DDS_CHECKSUM_BUILTIN64*, and *DDS_CHECKSUM_BUILTIN128* algorithms.

```
struct DDS_DomainParticipantQos dp_qos =
                                  DDS_DomainParticipantQos_INITIALIZER;

dp_qos.protocol.computed_crc_kind =  DDS_CHECKSUM_BUILTIN64;

dp_qos.protocol.allowed_crc_mask = DDS_CHECKSUM_BUILTIN32
                                 | DDS_CHECKSUM_BUILTIN64
                                 | DDS_CHECKSUM_BUILTIN128;
```

### 7.13.3 Participant Discovery and Participant Compatibility

*Connext Cert* ensures that participants establish communication with each other only when they have compatible checksum configurations. If *compute_crc* is *true*, all messages sent from the participant are protected by a checksum. Since each participant can use a different type of checksum, a mechanism is required to ensure that participants are compatible during discovery.

To bootstrap this mechanism, all participant announcements (if *compute_crc* is set to true) include a checksum of type *DDS_CHECKSUM_BUILTIN32*. The participant announcement carries information about the *computed_crc_kind* (the checksum kind used by the participant) and the *allowed_crc_mask* (the checksum kinds understood by the participant), and whether or not the participant requires a checksum for each RTPS message (if *require_crc* is set to true). Please note that messages with *DDS_CHECKSUM_BUILTIN32* checksum are *always* accepted to enable discovering new participants.

For a Participant (A) to match with another Participant (B), the *computed_crc_kind* of Participant (B) must be a strict subset of the *allowed_crc_mask* of Participant (A) and vice versa. If Participant (B) does not send a checksum (*compute_crc* is set to false), it can only match Participant (A) if it does not set *require_crc* to true.

### 7.13.4 Interoperability with Connext Professional

*Connext Professional* supports a CRC 32-bit checksum. However, the RTPS submessage used by *Connext Professional* to include a checksum is different from the one used by *Connext Cert* and what has been standardized by the OMG. *Connext Cert always* accepts *Connext Professional*'s CRC32 and treats it as a *DDS_CHECKSUM_BUILTIN32*. However, in order to enable interoperability with *Connext Professional* and enable *Connext Professional* to validate the checksum, it is necessary to change the transmit mode of *Connext Cert*. Two transmit modes are available:

- RTPS_CRC_TXMODE_OMG - Use the standard method as defined by the OMG. This is the default mode. The checksums sent by *Connext Cert* are *not* understood by *Connext Professional*, and *Connext Professional* will accept the messages as not having a CRC32.
- RTPS_CRC_TXMODE_RTICRC32 - CRC32 Mode. This mode sets the *computed_crc_kind* to *DDS_CRC_BUILTIN32*. The checksum sent by *Connext Cert* is understood by Pro. Use this option only if the *Connext Professional* application in your system needs checksum validation and has set *check_crc* to *true*.

## 7.14 Working With Sequences

### 7.14.1 Introduction

*RTI Connext Cert* 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.

### 7.14.2 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 Cert*:

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

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

| IDL Type | *Connext Cert* Type | *Connext Cert* 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 Cert must* be finite.

- All sequences defined in IDL are sized based on IDL properties and *must* not be resized. That is, *never* call **set_maximum()** on a sequence defined in IDL. This is particularly important for string sequences.
- Application defined sequences can be resized using **set_maximum()** or **ensure_length()**.
- There are two ways to use a **DDS_StringSeq** (they are type-compatible):
  - A **DDS_StringSeq** originating from IDL. This sequence is sized based on maximum sequence length *and* maximum string length.
  - A **DDS_StringSeq** originating from an application. In this case the sequence element memory is unmanaged.
- All sequences have an initial length of 0.

### Working with IDL Sequences

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

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

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

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

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

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

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

/* Assign the first 5 longs in long_seq */
for (i = 0; i < 5; ++i)
{
    *DDS_LongSeq_get_reference(&my_sample->long_seq,i) = i;
```

(continues on next page)

(continued from previous page)

```
    snprintf(*DDS_StringSeq_get_reference(&my_sample->string_seq,0),255,"SomeString %d",
↪i);
}

/* The delete call is _not_ available in Micro Cert */
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);
}

/* The delete call is _not_ available in Micro Cert */
SomeIdlTypeTypeSupport::delete_data(my_sample);
```

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

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

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

```
struct DDS_StringSeq app_seq = DDS_SEQUUENCE_INITIALIZER;

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

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

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

```
struct DDS_StringSeq app_seq = DDS_SEQUUENCE_INITIALIZER;

```

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

  ```
  struct DDS_StringSeq my_seq = DDS_SEQUENCE_INITIALIZER;

  DDS_StringSeq_ensure_length(&my_seq,10,20);

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

  /* The finalize call is _not_ available in Micro Cert */
  DDS_StringSeq_finalize(&my_seq);
  ```

## 7.15 Connext Cert Hardcoded Resource Limits

### 7.15.1 Introduction

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

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

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

### 7.15.2 Summary

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

### 7.15.3 Operating Services API (OSAPI)

- The maximum number of object IDs are 2^32-1
  - DDS objects require a unique object_id. The encoding dictated by the RTPS specification limits the number of DDS objects within a domain participant to 2^24.
  - User impact - None.
- The maximum number of timers that can be created is 8
  - Each DomainParticipant allocates 1 timer
    - User impact - The maximum number of domain participants in a single OS process is limited to 8. This limit is based on empirical data; only specialized applications such as tools typically use more than 2 domain participants.
- *Connext Cert* cannot run continously for longer than approximately 68 years.
  - User impact - Do not run an application using *Connext Cert* for longer than approximately 68 years before restarting it.
- *Connext Cert* does not support a calendar time after January 1 2038.
  - User impact
    - The DESTINATION_ORDER source_timestamp relies on the difference between two timestamps to determine if two samples are considered to have the same timestamp in case time has been adjusted backwards. A platform that relies on absolute time may not support this. It is possible to write samples with a manually specified timestamp to mitigate this limitation.

    * Timestamp information for samples may be incorrect after January 1 2038.

### 7.15.4 DDS C API

- Maximum Topic name length - 255 (including NUL termination)
  - The limit is specified as 256 including NUL termination in the RTPS specification, refer to 9.6.2.2.2 in the RTPS specification (OMG formal/2009-01-05).
- Maximum Type name length - 255 (including NUL termination)
  - The limit is specified as 256 including NUL termination in the RTPS specification, refer to 9.6.2.2.2 in the RTPS specification (OMG formal/2009-01-05).
- Maximum number of matched data-writers (per data-reader) - 1,000,000
  - This limit determines how many data-writers each data-reader can match.
- Maximum number of matched data-readers (per data-writer) - 100,000,000
  - This limit determines how many data-readers each data-writer can match.
- Maximum number of locators of each type which can be sent in the participant announcement - 4
  - This limit determines the number of unique network address that can be advertised as part of discovery. The limit is per locator type. That is, the limit is applicable to discovery and user-data (total of 4 each)
- Maximum number of discovery plugins which can be used by the domain participant - 1
  - User impact: Must choose either static or dynamic discovery.
- Maximum timeout for a DDS WaitSet is approximately 40 days.
  - **User impact: After 40 days, the DDS WaitSet will time out, possibly with** no active conditions.

### 7.15.5 Static Discovery Plugin (DPSE)

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

### 7.15.6 RTPS Protocol Implementation (RTPS)

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

## 7.16 Debugging

**Note:** Logging is *only* available in the *Debug* libraries.

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

For a list of error codes, please refer to Logging Reference.

### 7.16.1 Configuring logging

By default, *Connext Cert* sets the log verbosity to *Error* (see *Log message kinds* for details). The log verbosity can be changed at any time by calling OSAPI_Log_set_verbosity() using the desired verbosity as a parameter.

The *Connext Cert* log stores new log entries in a log buffer. You can set a custom buffer size with the OSAPI_Log_set_property() function. For example, to set a buffer size of 128 bytes:

```
struct OSAPI_LogProperty prop = OSAPI_LogProperty_INIITALIZER;

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

**Note:** If the buffer size is too small, log entries will be truncated in order to fit in the available buffer.

In order to provide *Connext Cert* with a mechanism for writing log messages, you have to set a function used for this purpose. This can be done via the function OSAPI_Log_set_property() as well. If your application code defines the function `my_log_write_buffer` like this:

```
#include <unistd.h>

static void
my_log_write_buffer(const char *buffer,RTI_SIZE_T length)
{
    write(1, buffer, length);
}
```

then you can set it as the `write_buffer` function for *Connext Cert* as follows:

```
struct OSAPI_LogProperty prop = OSAPI_LogProperty_INIITALIZER;

OSAPI_Log_get_property(&prop);
prop.write_buffer = my_log_write_buffer;
OSAPI_Log_set_property(&prop);
```

Alternatively, you 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. This allows you to redirect log entries to another logger, such as one native to a particular platform.

### 7.16.2 Log message kinds

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

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

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

### 7.16.3 Interpreting log messages and error codes

A log entry in *Connext Cert* 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. This ID is unique within a module.

  Although we refer to this as an "error" code, it exists for all log kinds (error, warning, info).

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

You can also configure *Connext Cert* to provide additional details per log message:

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

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

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

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

You will need to interpret the log message when an error or warning has occurred and its cause needs to be determined, or when you have set a log handler and are processing each log message based on its contents.

You can retrieve a description of an error code printed in a log message by following these steps:

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

# Chapter 8

# Working with RTI Connext Cert and RTI Connext

In some cases, it may be necessary to write an application that is compiled against both RTI *Connext Micro*, *RTI Connext Cert*, and *RTI Connext*. In general this is not easy to do because RTI *Connext Micro* and *RTI Connext Cert* supports a very limited set of features compared to *RTI Connext*. However, while *RTI Connext Cert* is a subset of RTI *Connext Micro*, it is relatively easy to write applications that support both.

Due to the nature of the DDS API and the philosophy of declaring behavior through QoS profiles instead of using different APIs, it may be possible to share common code. In particular, *RTI Connext* supports configuration through QoS profile files, which eases the job of writing portable code.

Please refer to *Introduction* for an overview of features and what is supported by *RTI Connext Cert*. Note that *RTI Connext* supports many extended APIs that are not covered by the DDS specification, for example APIs that create DDS entities based on QoS profiles.

## 8.1 Development Environment

There are no conflicts between *RTI Connext Cert* and *RTI Connext* with respect to library names, header files, etc. It is advisable to keep the two installations separate, which is the normal case.

*RTI Connext Cert* uses the environment variable RTIMEHOME to locate the root of the *RTI Connext Cert* installation.

*RTI Connext* uses the environment variable NDDSHOME to locate the root of the *RTI Connext* installation.

## 8.2 Non-standard APIs

The DDS specification omits many APIs and policies necessary to configure a DDS application, such as transport, discovery, memory, logging, etc. In general, *RTI Connext Cert* and *RTI Connext* do not share APIs for these functions.

It is recommended to configure *RTI Connext* using QoS profiles as much as possible.

## 8.3 QoS Policies

QoS policies defined by the DDS standard behave the same between *RTI Connext Cert* and *RTI Connext*. However, note that *RTI Connext Cert* does not always support all the values for a policy and in particular unlimited resources are not supported.

Unsupported QoS policies are the most likely reason for not being able to switch between *RTI Connext Cert* and *RTI Connext*.

## 8.4 Standard APIs

APIs that are defined by the standard behave the same between *RTI Connext Cert* and *RTI Connext*.

## 8.5 IDL Files

*RTI Connext Cert* and *RTI Connext* use the same IDL compiler (rtiddsgen) and *RTI Connext Cert* typically ships with the latest version. However, *RTI Connext Cert* and *RTI Connext* use different templates to generate code and it is not possible to share the generated code. Thus, while the same IDL can be used, the generated output must be saved in different locations.

## 8.6 Interoperability

In general, *RTI Connext Cert* and *RTI Connext* are wire interoperable, unless noted otherwise. Therefore, *RTI Connext Cert* is in general also compatible with RTI tools and other *RTI Connext* products. The following sections provide additional information for each product or module.

### 8.6.1 Discovery

When trying to establish communication between an *RTI Connext Cert* application that uses the Dynamic Participant / Static Endpoint (DPSE) discovery plugin and an RTI product based on *RTI Connext*, every participant in the DDS system must be configured with a unique participant name. While the static discovery functionality provided by *RTI Connext* allows participants on different hosts to share the same name, *RTI Connext Cert* requires every participant to have a different name to help keep the complexity of its implementation suitable for smaller targets.

Note that *RTI Connext Cert* only supports the DPSE discovery plugin. See the DPSE Examples Repository for information on how to configure *RTI Connext* Libraries, Tools, and Services to communicate with *RTI Connext Cert*.

### 8.6.2 Transports

When interoperating with *RTI Connext*, *RTI Connext Cert* must specify at least one unicast transport for each DataWriter and DataReader, either from DDS_DomainParticipantQos::transports or the endpoint DDS_DataReaderQos::transport and DDS_DataWriterQos::transport, since it expects to use the unicast transport's RTPS port mapping to determine automatic participant IDs if needed. This also affects *RTI Connext Cert* itself, where participant IDs must be set manually if only multicast transports are enabled.

Also, when interoperating with *RTI Connext*, only one multicast transport can be specified per *RTI Connext Cert* DataReader.

### 8.6.3 Admin Console

Admin Console can discover and display *RTI Connext Cert* applications that use full dynamic discovery (DPDE). When using static discovery (DPSE), it is required to use the Limited Bandwidth Endpoint Discovery (LBED) that is available as a separate product for *RTI Connext*. With the library a configuration file with the discovery configuration must be provided (just as in the case for products such as Routing Service, etc.). This is provided through the QoS XML file.

Data can be visualized from *RTI Connext Cert* DataWriters. Keep in mind that *RTI Connext Cert* does not currently distribute type information and the type information has to be provided through an XML file using the "Create Subscription" dialog. Unlike some other products, this information cannot be provided through the QoS XML file. To provide the data types to Admin Console, first run the code generator with the `-convertToXml` option:

```
rtiddsgen -convertToXml <file>
```

Then click on the "Load Data Types from XML file" hyperlink in the "Create Subscription" dialog and add the generated IDL file.

Other Features Supported:

- Match analysis is supported.
- Discovery-based QoS are shown.

The following resource-limits in *RTI Connext Cert* must be incremented as follows when using Admin Console:

- Add 24 to DDS_DomainParticipantResourceLimitsQosPolicy::remote_reader_allocation
- Add 24 to DDS_DomainParticipantResourceLimitsQosPolicy::remote_writer_allocation
- Add 1 to DDS_DomainParticipantResourceLimitsQosPolic::remote_participant_allocation
- Add 1 to DDS_DomainParticipantResourceLimitsQosPolicy::remote_participant_allocation if data-visualization is used

*RTI Connext Cert* does not currently support any administration capabilities or services, and does not match with the Admin Console DataReaders and DataWriters. However, if matching DataReaders and DataWriters are created, e.g., by the application, the following resource must be updated:

- Add 48 to DDS_DomainParticipantResourceLimitsQosPolicy::matching_writer_reader_pair_allocation

### 8.6.4 Distributed Logger

This product is not supported by *RTI Connext Cert*.

### 8.6.5 LabVIEW

The LabVIEW toolkit uses *RTI Connext*, and it must be configured as any other *RTI Connext* application. A possible option is to use the builtin *RTI Connext* profile: BuiltinQosLib::Generic.ConnextMicroCompatibility.

### 8.6.6 Monitor

This product is not supported by *RTI Connext Cert*.

### 8.6.7 Recording Service

#### RTI Recorder

RTI Recorder is compatible with *RTI Connext Cert* in the following ways:

- If static endpoint discovery is used, Recorder is compatible starting with version 5.1.0.3 and onwards.
- If dynamic endpoint discovery is used (not supported by *Connext Cert*), Recorder is compatible with *RTI Connext Cert* the same way it is with any other DDS application.
- In both cases, type information has to be provided via XML. Read Recording Data with RTI Connext Micro for more information.

#### RTI Replay

RTI Replay is compatible with *RTI Connext Cert* in the following ways:

- If static endpoint discovery is used, Replay is compatible starting with version 5.1.0.3 and onwards.
- If dynamic endpoint discovery is used (not supported by *Connext Cert*), Replay is compatible with *RTI Connext Cert* the same way it is with any other DDS application.
- In both cases, type information has to be provided via XML. Read Recording Data with RTI Connext Micro for more information on how to convert from IDL to XML.

#### RTI Converter

Databases recorded with *RTI Connext Cert* contains no type information in the DCPSPublication table, but the type information can be provided via XML. Read Recording Data with RTI Connext Micro for more information on how to convert from IDL to XML.

### 8.6.8 Wireshark

Wireshark fully supports *RTI Connext Cert*.

### 8.6.9 Persistence Service

*RTI Connext Cert* only supports VOLATILE and TRANSIENT_LOCAL durability and does not support the use of Persistence Service.

# Chapter 9

# API Reference

*RTI Connext Cert* 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

# Chapter 10

# Release Notes

## 10.1 Supported Platforms and Programming Languages

*Connext Cert* only supports C language bindings.

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

| OS | CPU | Compiler | RTI Architecture Abbreviation |
|---|---|---|---|
| QOS 2.2.1 (QNX® OS for Safety) | ARMv8 | qcc_gpp8.3.0 | armv8leElfqcc8.3.0CERT |

## 10.2 What's New in 2.4.15

The following features are new since *Connext Cert* 2.4.12.1.

### 10.2.1 Platform-independent code is now separate from OS and network stack integration

This release splits *Connext Cert* into two libraries:

- Platform Independent Libraries (PIL): binaries that support the basic features of *Connext Cert*. These are provided as precompiled binaries.
- Platform Support Libraries (PSL): binaries that support OS and network stack integration. These must be built from the provided source code.

This split ensures that the PIL does not need to be recertified for different platforms. Platform-specific PSLs can then be written for the same PIL without needing to recompile the *Connext Cert* code. Previous releases of *Connext Cert* were delivered as integrated libraries with the OS and network stack code included, which could not be changed without recompiling the entire library. See the *Library types* section for more information on this change.

Refer to *Building the PSL* for instructions on how to build the PSL from source.

### 10.2.2 Transfer large data samples quickly with Zero Copy v2

This release adds a new transport, Zero Copy v2, which can perform Zero Copy data transfer. Zero Copy transfer allows you to move large data samples without copying them, which increases throughput and reduces latency.

For more details, refer to *Zero Copy Transfer* and *Zero Copy v2 Transport*.

## 10.3 What's Fixed in 2.4.15

The following are fixes since *Connext Cert* 2.4.12.1.

[Critical]: System-stopping issue, such as a crash or data loss. [Major]: Significant issue with no easy workaround. [Minor]: Issue that usually has a workaround. [Trivial]: Small issue, such as a typo in a log.

### 10.3.1 [Critical] Failed to deliver samples when prior samples were lost

*Connext Cert* would sometimes fail to deliver samples to a *DataReader* if the last processed sample was a `HEARTBEAT` message that indicated lost samples.

[RTI Issue ID MICRO-7317]

### 10.3.2 [Critical] messageLength field was required in RTPS HEADER_EXTENSION for checksum calculations

Previously, *Connext Cert* required that the `messageLength` field be present in the RTPS `HEADER_EXTENSION` if a checksum was also included in the `HEADER_EXTENSION`. If the `messageLength` was not present, it was considered an error and the RTPS message was discarded.

Now, the `messageLength` field is optional. If it is not included, the `messageLength` will be assumed to be the size of the message payload.

*Connext Cert* and *Connext Professional* always include the `messageLength` in the `HEADER_EXTENSION` when a checksum is being sent. This fix will resolve compatibility when using checksums with other DDS implementations.

[RTI Issue ID MICRO-7172]

### 10.3.3 [Critical] DataReader on a Topic using an appendable type may have received samples with incorrect value

A *DataReader* subscribing to a *Topic* on an appendable type may have received incorrect samples from a matching *DataWriter*.

The problem only occurred when the *DataWriter* published a type with fewer members than the *DataReader* type. For example, consider a *DataWriter* on FooBase and a *DataReader* on FooDerived:

```
@appendable struct FooBase {
    sequence<uint8,1024>base_value;
};

@appendable struct FooDerived {
    sequence<uint8,1024> base_value;
    @default(12) uint8 derived_value;
};
```

In this case, the serialized sample stream would be padded with extra bytes to align the stream to 4 bytes as required by the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3. However, the additional padding bytes were incorrectly interpreted as part of the data and `derived_value` may have been set to a random value.

For example, in the case above when the *DataWriter* published a sample with type `FooBase`, in some cases the *DataReader* received a sample in which the field `derived_value` was set to 0 instead of 12.

**Note:** *Connext Cert* does not support the `@default` annotation.

[RTI Issue ID MICRO-6402]

### 10.3.4 [Critical] Failure to interoperate with other DDS implementations if default multicast locator was specified

*Connext Cert* did not interoperate with other DDS implementations when the default multicast locator was specified.

[RTI Issue ID MICRO-5148]

### 10.3.5 [Critical] Connext Cert may have repeated requesting a sample that was no longer available from a DataWriter

If *Connext Cert* detects a missing sample when using DDS_RELIABLE_RELIABILITY_QOS reliability, it will request the sample to be resent, but if the sample is no longer available from the *DataWriter*, the *DataWriter* may send a GAP message to indicate the sample is not longer available.

*Connext Cert* failed to interpret the GAP message correctly if the first sequence number in the GAP message was equal to the bitmap base of the GAP message. In this case, *Connext Cert* failed to ignore the no-longer-available sample and kept sending a request for the sample.

[RTI Issue ID MICRO-4668]

### 10.3.6 [Critical] DataWriter with BEST_EFFORT and TRANSIENT_LOCAL may have run out of resources

A *DataWriter* with `BEST_EFFORT` and `TRANSIENT_LOCAL` QoS policies may run out of resources when `DataWriterQos.resource_limits.max_samples_per_instance` > 1.

---

**Note:** Resending of historical samples (`DataWriterQos.durability.kind` = `TRANSIENT_LOCAL`) requires a `DataWriterQos.reliability.kind` = `RELIABLE` QoS policy. Thus, the combination of `BEST_EFFORT` and `TRANSIENT_LOCAL` is not useful, although it is a legal combination.

---

[RTI Issue ID MICRO-4508]

### 10.3.7 [Major] Code generation did not create proper type support for IDL files containing multiple Zero Copy types

*This issue only affected version 2.4.15 ZC ER-2.*

IDL files containing multiple types with the `@transfer_mode(SHMEM_REF)` annotation would only generate type support functions for the first type in the file.

[RTI Issue ID MICRO-8461]

### 10.3.8 [Major] Samples with meta-information were not delivered to the user if they arrived when history cache was full

When a *DataReader*'s history cache was full, samples containing meta-information from matched *DataWriters* were not delivered to the user.

[RTI Issue ID MICRO-8063]

### 10.3.9 [Major] Possible segmentation fault when a Zero Copy-enabled DataReader called read() after unmatching with a DataWriter

*This issue only affected version 2.4.15 ZC ER-2.*

If a Zero Copy-enabled *DataReader* attempted to retrieve any samples it previously received from an unmatched *DataReader* by calling `read()`, a segmentation fault could occur.

[RTI Issue ID MICRO-7880]

### 10.3.10 [Major] Incorrect deserialization of CDR encapsulation padding bits

The padding bytes in a sample were deserialized incorrectly. This resulted in samples being dropped if the number of padding bytes in a sample was greater than zero.

[RTI Issue ID MICRO-6799]

### 10.3.11 [Major] Heap memory was allocated by the Zero Copy v2 transport during DDS discovery

*This issue only affected version 2.4.15 ZC ER-2.*

The Zero Copy v2 transport performed heap memory allocations at runtime, specifically at the time DDS discovery occurred. This was out of compliance with CERT criteria.

Now, all heap memory required by the Zero Copy v2 transport is allocated at initialization time (the point when all DDS objects are created by user code), and no further allocations are performed.

[RTI Issue ID MICRO-5813]

### 10.3.12 [Minor] Incorrect heartbeat sent before first sample when first_write_sequence_number is not 1

If the `DataWriterQos.protocol.rtps_reliable_writer.first_write_sequence_number` was different from the default value 1, heartbeats sent before the first sample was written would indicate 1 as the first sample available. This caused *DataReaders* to wait for samples with a sequence number less than `DataWriterQos.protocol.rtps_reliable_writer.first_write_sequence_number` until a heartbeat with the correct first sequence number was received.

[RTI Issue ID MICRO-4081]

### 10.3.13 [Minor] DDS_String_alloc and related functions failed when passed 0 as length

Passing a `length` of `0` to DDS_String_alloc returned NULL. Now, you can pass a `length` of `0` to DDS_String_alloc and get a valid return value (one allocated byte with a null terminator). This fix also affects other functions that utilize DDS_String_alloc.

[RTI Issue ID MICRO-5452]

### 10.3.14 [Trivial] DDS_Wstring_cmp did not match the implementation name DDS_Wstring_compare

The DDS Wstring compare function was incorrectly documented as being `DDS_Wstring_cmp` instead of `DDS_Wstring_compare`.

[RTI Issue ID MICRO-3529]

## 10.4 Previous Releases

### 10.4.1 What's New in 2.4.12.1

#### Support for AUTOSAR Classic

This release includes support for Elektrobit AUTOSAR 4.0.3 and Mentor™ AUTOSAR 4.2.2 on Infineon AURIX TriCore TC297.

**Warning:** AUTOSAR Classic is not supported in version 2.4.15 ZC ER-2.

#### Support for detecting corrupted RTPS messages

This release includes support for detecting and discarding corrupted RTPS messages. A checksum is computed over the DDS RTPS message including the RTPS Header. This checksum is sent as a new RTPS submessage. The subscribing application detects this new submessage and validates the contained checksum. When a corrupted RTPS message is detected, the message is dropped.

To enable the use of a checksum in a *DomainParticipant*, there are three new fields in the *WireProtocolQosPolicy*: *compute_crc*, *check_crc*, and *require_crc*:

- To send the checksum, enable *compute_crc* at the sending application.
- To drop corrupted messages, enable *check_crc*.

- To ignore a participant with compute_crc set to *false*, enable *require_crc*.

Please refer to *Message Integrity Checking* in the *Connext Cert* User's Manual for details.

#### Reduced default socket send/receive buffer size for QNX systems

Some QNX kernels have a maximum send and receive socket buffer size smaller than the default value used by *Connext Cert*. The default send and receive socket buffer size has been changed to 64 Kbytes in *Connext Cert* for QNX builds.

### 10.4.2 What's Fixed in 2.4.12.1

#### *PUBLICATION_MATCHED_STATUS* and *SUBSCRIPTION_MATCHED_STATUS* may never have triggered a WaitSet if the status was enabled after the *DomainParticipant* was enabled

A StatusCondition with *PUBLICATION_MATCHED_STATUS* or *SUBSCRIPTION_MATCHED_STATUS* enabled may never have triggered a WaitSet, if the status was enabled after the *DomainParticipant* was enabled.

This issue has been resolved.

[RTI Issue ID MICRO-2219]

#### A RTPS *max_window_size* not divisible by 32 may have resulted in retransmission of wrong sequence number

An RTPS *max_window_size* not divisible by 32 may have caused retransmission of a sequence number not being requested. Note that the default value is divisible by 32.

This issue has been resolved.

[RTI Issue ID MICRO-2287]

#### POSIX mutex implementation did not conform with FACE Safety Profile

The POSIX mutex implementation did not conform with the FACE Safety Profile. This release conforms to the FACE Safety profile for single-core CPU architectures.

[RTI Issue ID MICRO-2275]

#### Waitset with timeout of 0 did not return immediately

A Waitset with a 0 timeout did not return immediately, but was rounded up to one clock period.

This issue has been resolved.

[RTI Issue ID MICRO-2278, MICRO-2264]

#### Invalid samples in batched data did not count as 'lost samples'

Invalid samples in batched data were not counted as lost samples, and did not trigger *Connext Cert* to call *on_sample_lost()* when the "on_sample_lost" notification was enabled.

This issue has been resolved.

[RTI Issue ID MICRO-2289]

#### Unicast DataReader stopped receiving samples after DataWriter matched with a multicast DataReader

A *DataReader* with a unicast locator stopped receiving samples from a matched *DataWriter* when another *DataReader* with a multicast locator matched with that *DataWriter*.

This problem has been resolved. Now all matched *DataReaders* will receive samples, regardless of whether their locators are unicast or multicast.

[RTI Issue ID MICRO-2369]

#### DomainParticipant was not rediscovered if it restarted before expiring

If a *DomainParticipant* was terminated and restarted before its lease duration expired, it would not be successfully rediscovered. For example, if an application with a *DomainParticipant* was terminated with Control-C and restarted before the *DomainParticipant's* lease duration expired, the *DomainParticipant* would not be rediscovered. However, if the *DomainParticipant* was deleted with *delete_participant()* this problem would not occur.

This issue has been resolved.

[RTI Issue ID MICRO-2672]

#### Unexpected behavior when copying a DDS_UnsignedShortSeq with 0 length

When copying a *DDS_UnsignedShortSeq* with 0 length, the destination sequence length was not set to 0.

This issue has been resolved.

[RTI Issue ID MICRO-2756]

#### Local variables in header file may have caused compiler warning

Local variables were incorrectly defined in ReaderHistory.c and may have caused a compiler warning.

This issue has been resolved.

[RTI Issue ID MICRO-2785]

#### Non-default timer resolutions may have caused an incorrect timeout

Compiling *Connext Cert* with a non-default timer resolution may have caused incorrect timeouts.

This issue has been resolved.

[RTI Issue ID MICRO-2794]

#### Missing checks for *max_routes_per_reader* and *max_routes_per_writer*

The *DDS_DataReaderQos.reader_resource_limits.max_routes_per_writer* and *DDS_DataWriterQos.writer_resource_limits.max_routes_per_reader* were missing a check that the values were in the range [1,2000]. They were also missing from the methods *DDS_DataReaderQos_is_equal* and *DDS_DataWriterQos_is_equal* respectively.

This issue has been resolved.

[RTI Issue ID MICRO-2830, MICRO-2937]

#### Missing NULL checks for *enabled_transports*

In previous releases, it was not checked that the *enabled_transports* QoS policy setting did not contain NULL pointers.

This issue has been resolved.

[RTI Issue ID MICRO-3117]

#### Possible exception due to misaligned RTPS header

In previous releases, if multiple RTPS messages were received in the same UDP payload, a misaligned RTPS message header could cause an exception.

**Note:** *RTI Connext Cert* does not send multiple RTPS messages in the same UDP payload.

This issue has been resolved.

[RTI Issue ID MICRO-2866]

#### ***DDS_SubscriptionBuiltinTopicData_copy*** did not copy the PresentationQosPolicy

The *DDS_SubscriptionBuiltinTopicData_copy* function did not copy the PresentationQosPolicy.

This issue has been resolved.

[RTI Issue ID MICRO-2897]

#### Possible failure to start timer

On architctures using the *posix* port of *Connext Cert*, the DomainParticipantFactory may have failed to initialize if compiled to use signals or if CLOCK_MONOTONIC was not available.

This issue has been resolved.

[RTI Issue ID MICRO-2904]

#### Sample timestamp now set to 0 if timestamp cannot be retrieved

If the reception timestamp for a sample cannot be retrieved, the reception timestamp is set to 0.

[RTI Issue ID MICRO-2909]

#### ***Qos_copy*** functions did not validate input arguments

In previous releases, the *Qos_copy* APIs did not validate that the input arguments were not NULL.

This issue has been resolved.

[RTI Issue ID MICRO-2913]

#### Unused parameter ***DOMAIN_PARTICIPANT_RESOURCE_LIMITS.matching_reader_writer_pair_allocation*** removed

The QoS policy setting *DOMAIN_PARTICIPANT_RESOURCE_LIMITS.matching_reader_writer_pair_allocation* was not used and has been removed from the *DOMAIN_PARTICIPANT_RESOURCE_LIMITS* structure.

[RTI Issue ID MICRO-2915]

### *DDS_DomainParticpant_add_peer* may have returned success on failure

*DDS_DomainParticpant_add_peer* may have returned success even if the peer was not added.

This issue has been resolved.

[RTI Issue ID MICRO-2929]

### *DDS_StringSeq_copy* did not validate input arguments

In previous versions, *DDS_StringSeq_copy* did not check that the source and destination arguments were different before copying.

This issue has been resolved.

In addition, the documentation for *Seq_copy* has been updated to clearly state that overlapping memory regions are not supported, with the exception of copying to itself.

[RTI Issue ID MICRO-2964]

### Possible NULL pointer exception in generated code if the system was out of memory

In previous releases, it was possible to get a NULL pointer exception in the generated code if the system was out of memory during initialization.

This issue would have occurred during DDS entity creation, as memory is only allocated during entity creation.

This issue has been resolved.

[RTI Issue ID MICRO-2986]

### A DataWriter could run out of resources if sample was not added to cache

In rare cases, a *DataWriter* could run out of resources if a sample could be successfully serialized, but not added to the writer cache.

This issue has been resolved.

[RTI Issue ID MICRO-3034]

### Possible serialization beyond stream buffer

In previous releases, *CDR_Stream_check_size* did not check for underflow. As a result, it was possible to serialize data beyond the buffer boundary if the buffer assigned to the stream was too small.

This is only an issue for applications assigning too small of a buffer to a stream.

This issue has been resolved.

[RTI Issue ID MICRO 3147, MICRO-3200]

#### *RELIABILITY.max_blocking_time* must be zero

In previous releases, a non-zero *RELIABILITY.max_blocking_time* was supported on a *DataReader*. This feature is not supported in this release.

[RTI Issue ID MICRO-3148]

#### Possible DataReader or DataWriter creation failure with multiple DomainParticipants

In previous releases, creating *DataReaders* or *DataWriters* in different threads for different *DomainParticipants* could fail due to a race condition.

This issue has been resolved.

[RTI Issue ID MICRO-3151]

#### Incorrect *lease_duration* may have been used for a discovered participant

In previous releases, if the *lease_duration* was not sent by a remote *DomainParticipant*, a previously received value was used instead.

This issue has been resolved.

Note that RTI's DDS implementations send the *lease_duration*.

[RTI Issue ID MICRO-3254]

#### Missing consistency check for *DESTINATION_ORDER.source_timestamp_tolerance*

In previous releases, a check that *DESTINATION_ORDER.source_timestamp_tolerance* was normalized was missing (nanosecond < 1 seconds).

This issue has been resolved.

[RTI Issue ID MICRO-3272]

#### Improved error detection for unresolved addresses

In previous releases, an unresolved address was ignored. In this release, if an address cannot be resolved, it results in a failure. This means that all addresses passed to the *add_peer* API and the *enabled_transports* QoS policy must be valid, otherwise entity creation will fail.

[RTI Issue ID MICRO-3276]

#### *DDS_StatusCondition_set_enabled_statuses* did not trigger if an active condition was enabled

In previous releases, if a StatusCondition enabled by a call to *DDS_StatusCondition_set_enabled_statuses* was already active, the StatusCondition did not trigger.

This issue has been resolved.

[RTI Issue ID MICRO-3308]

#### Race condition in DDS *enable* APIs

In previouses releases, a race condition existed if the same DDS entity was enabled from multiple threads at the same time.

This issue has been resolved.

[RTI Issue ID MICRO-3311]

#### *disable_auto_interface_config* is not available in *Connext Cert*

In previous versions of *Connext Cert*, *disable_auto_interface_config* was available in the *UDP_InterfaceFactoryProperty* structure, but was ignored (always TRUE). This variable is no longer available in *Connext Cert*.

[RTI Issue ID MICRO-3322]

#### DDS WaitSet may have timed out later than timeout value

In very rare cases, an error message taking a mutex may have been logged when using the POSIX real-time timers. This may have resulted in a delayed timeout for *DDS_WaitSets*.

This issue has been resolved.

[RTI Issue ID MICRO-3330]

## 10.5 Known Issues

### 10.5.1 Logging mechanism is not included by default

*Connext Cert* supports logging in its Debug libraries, but the Platform Support Library (PSL) does not include a default logging mechanism. To see logging, you must set a log handler or log display function in your application.

For more information, see *Debugging*.

[RTI Issue ID MICRO-8534]

### 10.5.2 Resources are not freed when calling unregister_instance on a reliable Zero Copy DataWriter

When using a reliable Zero Copy *DataWriter*, calling FooDataWriter_unregister_instance() does not return the instance's resources to the appropriate pool for reuse.

As a workaround for this issue, you should size the DataWriterQos.resource_limits.max_instances parameter to accommodate the total number of instances that will be used during the lifetime of the *DataWriter*. This issue is not present when using a `DataWriterQos.reliability.kind` of DDS_BEST_EFFORT_RELIABILITY_QOS.

[RTI Issue ID MICRO-7253]

### 10.5.3 Multiple endpoints not matched properly when using Zero Copy

When attempting to match *DataWriters* and *DataReaders* that are created in already-enabled *DomainParticipants*, the matching may fail.

This issue can be worked around by creating all endpoints in a *DomainParticipant* before enabling the *DomainParticipant*.

[RTI Issue ID MICRO-5790]

### 10.5.4 Cannot set consistency mode for SharedQWriter to anything but ROBUST

Attempting to call `NETIO_ZCOPY_SharedQReader_open` or `NETIO_ZCOPY_SharedQReader_open_by_name` on a SharedQReader initialized for consistency mode results in an error if the SharedQ was initialized with a consistency mode other than *ROBUST*. There is currently no known workaround for this issue.

[RTI Issue ID MICRO-5784]

### 10.5.5 VOLATILE Zero Copy DataReaders behave as TRANSIENT_LOCAL

When a *DataReader* matches a *DataWriter* via Zero Copy communication, it has visibility into all samples that are present at that time, independent of its setting for the `DURABILITY` QoS. This may be unexpected and is not consistent with *VOLATILE DataReaders* that match to that same *DataWriter* via other transports.

[RTI Issue ID MICRO-5550]

### 10.5.6 Samples cannot be reclaimed if Zero Copy application does not return loan

If a *DomainParticipant* subscribing to Zero Copy data dies after reading or taking Zero Copy samples but before returning their loan, or never returns the loan for some other reason, then those samples can never be reclaimed for subsequent publishing and can be considered lost resources.

There is currently no known workaround that entirely removes the risk of samples becoming unavailable for reclaiming. The impact of such a scenario can be decreased by following these recommendations:

- On the receiving side, process samples one at a time, by setting the `max_samples` parameter to 1 when invoking the `read()` or `take()` family of functions. This will limit the impact of a dying process to a resource loss of 1 sample at most.
- Increase the `max_samples` resource limit on the *DataWriter*, to exceed the number of expected matching *DataReaders*. If the previous recommendation is followed as well, then the sample resources might decrease due to subscribing process failures, but never be exhausted.

[RTI Issue ID MICRO-5834]

### 10.5.7 Maximum number of components limited to 9

The maximum number of components that can be registered by an application is limited to 9, or 10 if the *Zero Copy v2 Transport* is not enabled.

[RTI Issue ID MICRO-3571]

### 10.5.8 Static endpoint discovery requires unique object IDs across all remote endpoints

When using static endpoint discovery (DPSE), *Connext Cert* requires that the `object_id` for statically asserted remote endpoints must be unique across all remote endpoints, as opposed to just among remote endpoints within the same *DomainParticipant*.

[RTI Issue ID MICRO-211]

# Chapter 11

# 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 12

# Third-Party and Open Source Software

This section outlines Real-Time Innovations (RTI) usage of first-level third-party open source software in the *RTI Connext Cert* libraries and utilities.

## 12.1 Connext Cert Libraries

### 12.1.1 crc32c.c

- Related to: RTPS CRC-32 checksum support
- Version 1.1
- Third-Party Software License:

(continues on next page)

(continued from previous page)

### 12.1.2 MD5

- Related to: DDS keys implementation, content-filtered topics (to sign the filter), persistence service (to generate writer-side unique identification), Integration Toolkit for AUTOSAR (DDS-IDL Service Interface code generation)
- Software is included in core DDS middleware libraries, in the Connext DDS Micro libraries, in the Connext DDS Cert libraries and in the Integration Toolkit for AUTOSAR.
- Third-Party Software License:

## 12.2 Third-Party Software used by the RTIDDSGEN Code-Generation Utility

### 12.2.1 ANTLR

- This software is distributed with rtiddsgen (RTI Code Generator) as a jar file. The source code is not modified or shipped. In addition, the output produced by this software from a grammar file is part of the rtiddsgen JAR file. This has a dependency on ANTLR Runtime, StringTemplates v.3.2.1 and ST4 v.4.0.4.
- Version: Release 3.5.2
- Open Source Software License: https://www.antlr3.org/license.html

### 12.2.2 Apache Commons Lang

- Used on Code Generator. We only use the class StringUtils.
- Version 2.6
- Open Source Software License: Apache License Version 2.0 (full text found in the Appendix).

### 12.2.3 Apache Log4j 2

- This software is distributed with rtiddsgen (RTI Code Generator) as a jar file. The source code is not modified or shipped.
- Source: https://logging.apache.org/log4j/2.12.x/license.html
- Version: 2.17.1

### 12.2.4 Apache Velocity

- This software is included in rtiddsgen (RTI Code Generator). The source code is not modified or shipped.
- Source: http://velocity.apache.org/
- Version: 2.3
- Open Source Software License: The Apache Software License, Version 2.0
  http://velocity.apache.org/engine/devel/license.html

### 12.2.5 AdoptOpenJDK JRE

- Version 17.0.6 (LTS) Hotspot JVM
- The JRE binaries of the software are distributed with RTI Connext software that uses rtiddsgen (RTI Code Generator). The source code is not modified or shipped.
- https://adoptopenjdk.net/about.html
- Open Source Software Licenses:

Build scripts and other code to produce the binaries, the website and other build infrastructure are licensed under Apache License, Version 2.0. See Appendix. OpenJDK code itself is licensed under GPL v2 with Classpath Exception (GPLv2+CE). See below, in this section.

For Open JDK:

```
The GNU General Public License (GPL)

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

### 12.2.6 Gson

- Version 2.9.1
- Portions of rtiddsgen (RTI Code Generator) are built using Gson.
- Open Source Software License: The Apache Software License, Version 2.0 (full text found in the *Appendix*)

## 12.3 Appendix – Open Source Software Licenses

### 12.3.1 Apache License version 2.0, January 2004 (http://www.apache.org/licenses/)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

### 12.3.2 GNU GENERAL PUBLIC LICENSE Version 2, June 1991

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
    appropriate copyright notice and a notice that there is no warranty (or
    else, saying that you provide a warranty) and that users may redistribute
    the program under these conditions, and telling the user how to view a copy
    of this License.  (Exception: if the Program itself is interactive but does
    not normally print such an announcement, your work based on the Program is
    not required to print an announcement.)

These requirements apply to the modified work as a whole.  If identifiable
sections of that work are not derived from the Program, and can be reasonably
considered independent and separate works in themselves, then this License, and
its terms, do not apply to those sections when you distribute them as separate
works.  But when you distribute the same sections as part of a whole which is a
work based on the Program, the distribution of the whole must be on the terms
of this License, whose permissions for other licensees extend to the entire
whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your
rights to work written entirely by you; rather, the intent is to exercise the
right to control the distribution of derivative or collective works based on
the Program.

In addition, mere aggregation of another work not based on the Program with the
Program (or with a work based on the Program) on a volume of a storage or
distribution medium does not bring the other work under the scope of this
License.

3. You may copy and distribute the Program (or a work based on it, under
Section 2) in object code or executable form under the terms of Sections 1 and
2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable source
    code, which must be distributed under the terms of Sections 1 and 2 above
    on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three years, to
    give any third party, for a charge no more than your cost of physically
    performing source distribution, a complete machine-readable copy of the
    corresponding source code, to be distributed under the terms of Sections 1
    and 2 above on a medium customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer to
    distribute corresponding source code.  (This alternative is allowed only
    for noncommercial distribution and only if you received the program in
    object code or executable form with such an offer, in accord with
    Subsection b above.)

The source code for a work means the preferred form of the work for making
modifications to it.  For an executable work, complete source code means all
the source code for all modules it contains, plus any associated interface
definition files, plus the scripts used to control compilation and installation
of the executable.  However, as a special exception, the source code
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

# Chapter 13

# Contact Support

We welcome your input on how to improve *RTI Connext Cert* 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 14

# 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.