

RTI Connext Cert

User's Manual

Version 2.4.16.1



Contents

1	Introduction	2
1.1	What is RTI Connex Cert?	2
1.1.1	Publish-Subscribe Middleware	3
1.2	Supported DDS Features	3
1.2.1	DDS Entity Support	3
1.2.2	DDS QoS Policy Support	3
1.3	Standards and Interoperability	4
1.3.1	DDS Wire Compatibility	5
1.3.2	Profile / Feature	5
1.3.3	DDS API Support	7
1.4	RTI Connex Documentation	7
1.5	OMG DDS Specification	7
1.6	Other Products	7
2	Installation	9
2.1	Installing the RTI Connex Cert Package	9
2.2	Overview of the Host Bundle	10
2.3	Overview of the Target Bundle	11
2.3.1	Library types	11
2.3.2	Library descriptions	12
2.4	Directory Structure	13
3	Building Connex Cert	14
3.1	Setting up the build environment	15
3.1.1	The host environment	15
3.1.2	The target environment	16
3.2	Building the PSL	16
3.2.1	Building the PSL with rtime-make	16
3.2.2	Building the PSL with CMake	17
	Preparing to build	17
	Creating build files from the command line	18
	CMake flags used by Connex Cert	18
3.3	Compile-time options	19
3.3.1	Debug information	19
3.3.2	Platform selection	20
3.3.3	Compiler selection	20
3.3.4	UDP options	21

3.4	Custom build environments	21
3.4.1	Importing the Connexrt Cert code	21
4	Platform Notes	23
4.1	Introduction	23
4.1.1	Library types	23
4.1.2	Supported libraries by platform	24
4.1.3	Supported transports by platform	24
4.2	QNX Platforms	25
4.2.1	How the PIL was built for QNX platforms	25
4.2.2	Building the PSL from source for QNX platforms	26
4.2.3	Building QNX applications with Connexrt Cert	26
5	Getting Started	27
5.1	GitHub Example	27
5.2	Generating Examples	27
5.2.1	Default example	28
5.2.2	Custom example	28
5.2.3	Descriptions of generated examples	28
5.2.4	How to compile the generated examples	29
5.2.5	How to run the generated examples	29
6	Developing Applications	31
6.1	Prepare Your Development Environment	31
6.1.1	Set environment variables	31
6.1.2	Add required preprocessor flags	31
6.1.3	Link applications and libraries	32
6.2	Define a Data Type	32
6.3	Generate Type Support Code with rtiddsgen	33
6.4	Create an Application	34
6.4.1	Registry Configuration	34
6.5	Configure UDP Transport	36
6.6	Create DomainParticipant, Topic, and Type	37
6.6.1	Register Type	38
6.6.2	Create Topic of Registered Type	38
6.6.3	DPSE Discovery: Assert Remote Participant	39
6.7	Create Publisher	39
6.8	Create DataWriter	40
6.8.1	DPSE Discovery: Assert Remote Subscription	41
6.8.2	Writing Samples	41
6.9	Create Subscriber	42
6.10	Create DataReader	42
6.10.1	DPSE Discovery: Assert Remote Publication	44
6.10.2	Receiving Samples	45
6.10.3	Filtering Samples	47
7	User's Manual	49
7.1	Initializing the Connexrt Cert Library	49

7.1.1	rtiddsgen	50
7.1.2	The Connex Cert System API	51
7.1.3	Component Registration	51
7.2	Data Types	52
7.2.1	Introduction to the Type System	54
	Sequences	55
	Strings and Wide Strings	55
7.2.2	Creating User Data Types with IDL	57
7.2.3	Working with DDS Data Samples	58
7.3	DDS Entities	58
7.4	Sending Data	59
7.4.1	Preview: Steps to Sending Data	59
7.4.2	Publishers	60
7.4.3	DataWriters	61
7.4.4	Publisher QosPolicies	61
7.4.5	DataWriter QosPolicies	61
7.5	Receiving Data	61
7.5.1	Preview: Steps to Receiving Data	61
7.5.2	Subscribers	63
7.5.3	DataReaders	64
7.5.4	Using DataReaders to Access Data (Read & Take)	64
7.5.5	Subscriber QosPolicies	64
7.5.6	DataReader QosPolicies	64
7.6	DDS Domains	65
7.6.1	Fundamentals of DDS Domains and DomainParticipants	65
7.6.2	Discovery Announcements	66
7.7	Transports	67
7.7.1	Introduction	67
7.7.2	Transport Limits	68
	IDL Data Types and Size	68
	Maximum Transmission Unit (MTU)	68
	Maximum Receive Unit (MRU)	69
7.7.3	Transport Registration	69
7.7.4	Transport Addresses	70
	Reserving Addresses and Ports	71
	Address Limitations	71
	Address Notation	71
7.7.5	RTPS	72
	Registration of RTPS	72
	Overriding the Builtin RTPS Checksum Functions	73
	Example	74
7.7.6	INTRA Transport	77
	Registering the INTRA Transport	77
	Reliability and Durability	77
	Threading Model	78
7.7.7	NETIO Datagram Transport	78
	Registering a Datagram interface	78
	Addressing a Datagram transport	80

	Setting up the Datagram UDP	81
	User interface	83
7.7.8	Zero Copy v2 Transport	83
	Generate type support files	84
	Initialize the Zero Copy v2 transport	84
	Register the Zero Copy v2 transport	85
	Enable transports	86
	Manage DDS samples	86
	Configuration	87
	Using multiple Zero Copy v2 transport instances	90
7.8	Discovery	90
7.8.1	What is Discovery?	90
	Simple Participant Discovery	91
7.8.2	Configuring Participant Discovery Peers	91
	The Peer Address	92
7.8.3	Configuring Initial Peers and Adding Peers	93
7.8.4	Configuring Discovery Data Reception	93
7.8.5	Configuring User Data Reception	95
7.8.6	Configuring Enabled Transports per DataReader or DataWriter	96
7.8.7	Discovery Plugins	96
	Static Discovery Plugin	96
7.8.8	Asymmetric Matching and Lost Samples	99
7.8.9	DomainParticipant Discovery by Name	99
7.9	Generating Type Support with rtiddsgen	100
7.9.1	Why Use rtiddsgen?	100
7.9.2	IDL Type Definition	100
7.9.3	Generating Type Support	100
	C	101
	Notes on Command-Line Options	101
	Generated Type Support Files	101
7.9.4	Using custom data-types in Connexrt Cert Applications	102
7.9.5	Customizing generated code	102
7.9.6	Unsupported Features of rtiddsgen with Connexrt Cert	103
7.10	Thread Safety	103
7.10.1	Calling DDS APIs from listeners and callbacks	103
7.10.2	Calling DDS APIs from a type-plugin	104
7.11	Memory Model	104
7.11.1	Introduction	104
7.11.2	Dynamic Memory Allocation	105
7.11.3	Resource Limits	105
7.12	Zero Copy Transfer	107
7.12.1	Compatibility	107
7.12.2	Overview	108
7.12.3	Getting started	108
	Writing samples	109
	Reading samples	109
7.12.4	Synchronizing samples	110
7.12.5	Caveats	111

7.13	Message Integrity Checking	111
7.13.1	RTPS Checksum	112
7.13.2	Configurations	112
	Selecting a checksum algorithm	112
	Configuring the DDS DomainParticipant	113
7.13.3	Participant Discovery and Participant Compatibility	114
7.13.4	Interoperability with Connex Professional	114
7.14	Working With Sequences	115
7.14.1	Introduction	115
7.14.2	Working with Sequences	115
	Overview	115
	Working with IDL Sequences	116
	Working with Application Defined Sequences	118
7.15	Connex Cert Hardcoded Resource Limits	119
7.15.1	Introduction	119
7.15.2	Summary	119
7.15.3	Operating Services API (OSAPI)	119
7.15.4	DDS C API	120
7.15.5	Static Discovery Plugin (DPSE)	120
7.15.6	RTPS Protocol Implementation (RTPS)	121
7.16	Debugging	121
7.16.1	Configuring logging	121
7.16.2	Log message kinds	122
7.16.3	Interpreting log messages and error codes	122
8	Working with RTI Connex Cert and RTI Connex	124
8.1	Development Environment	124
8.2	Non-standard APIs	125
8.3	QoS Policies	125
8.4	Standard APIs	125
8.5	IDL Files	125
8.6	Interoperability	125
8.6.1	Discovery	126
8.6.2	Transports	126
8.6.3	Admin Console	126
8.6.4	Distributed Logger	127
8.6.5	LabVIEW	127
8.6.6	Monitor	127
8.6.7	Recording Service	127
	RTI Recorder	127
	RTI Replay	128
	RTI Converter	128
8.6.8	Wireshark	128
8.6.9	Persistence Service	128
9	API Reference	129
10	Release Notes	130

10.1	Supported Platforms and Programming Languages	130
10.2	What's New in 2.4.16.1	130
10.2.1	Improve system robustness with DomainParticipant rediscovery	131
10.2.2	Added support for QOS 8.0	131
10.3	What's Fixed in 2.4.16.1	131
10.3.1	[Critical] When using Zero Copy, the sample returned by <code>get_loan()</code> could have an insufficient size	131
10.3.2	[Major] Incorrect check for multicast address	132
10.3.3	[Major] <code>udpv4_singlethr_init_port_entry()</code> function caused UDP communication to fail	132
10.4	Previous Releases	132
10.4.1	What's New in 2.4.16	132
	Deprecations and Removals	132
10.4.2	What's Fixed in 2.4.16	133
	Discovery	133
	Reliability Protocol and Wire Representation	134
	Generated Code (C)	134
	APIs (C)	134
	Memory Leaks/Growth	135
	Data Corruption	135
	Interoperability	135
	Other	136
10.4.3	What's New in 2.4.15	137
	Platform-independent code is now separate from OS and network stack integration	137
	Transfer large data samples quickly with Zero Copy v2	138
10.4.4	What's Fixed in 2.4.15	138
	[Critical] Failed to deliver samples when prior samples were lost	138
	[Critical] <code>messageLength</code> field was required in RTPS <code>HEADER_EXTENSION</code> for checksum calculations	138
	[Critical] <code>DataReader</code> on a Topic using an appendable type may have received samples with incorrect value	139
	[Critical] Failure to interoperate with other DDS implementations if default multicast locator was specified	139
	[Critical] Connexrt Cert may have repeated requesting a sample that was no longer available from a <code>DataWriter</code>	140
	[Critical] <code>DataWriter</code> with <code>BEST_EFFORT</code> and <code>TRANSIENT_LOCAL</code> may have run out of resources	140
	[Major] Code generation did not create proper type support for IDL files containing multiple Zero Copy types	140
	[Major] Samples with meta-information were not delivered to the user if they arrived when history cache was full	140
	[Major] Possible segmentation fault when a Zero Copy-enabled <code>DataReader</code> called <code>read()</code> after unmatched with a <code>DataWriter</code>	141
	[Major] Incorrect deserialization of CDR encapsulation padding bits	141
	[Major] Heap memory was allocated by the Zero Copy v2 transport during DDS discovery	141

[Minor]	Incorrect heartbeat sent before first sample when <code>first_write_sequence_number</code> is not 1	141
[Minor]	<code>DDS_String_alloc</code> and related functions failed when passed 0 as length	142
[Trivial]	<code>DDS_Wstring_cmp</code> did not match the implementation name <code>DDS_Wstring_compare</code>	142
10.4.5	What's New in 2.4.12.1	142
	Support for AUTOSAR Classic	142
	Support for detecting corrupted RTPS messages	142
	Reduced default socket send/receive buffer size for QNX systems	143
10.4.6	What's Fixed in 2.4.12.1	143
	<code>PUBLICATION_MATCHED_STATUS</code> and <code>SUBSCRIPTION_MATCHED_STATUS</code> may never have triggered a <code>WaitSet</code> if the status was enabled after the <i>DomainParticipant</i> was enabled	143
	A RTPS <i>max_window_size</i> not divisible by 32 may have resulted in retransmission of wrong sequence number	143
	POSIX mutex implementation did not conform with FACE Safety Profile	143
	<code>Waitset</code> with timeout of 0 did not return immediately	143
	Invalid samples in batched data did not count as 'lost samples'	144
	Unicast <code>DataReader</code> stopped receiving samples after <code>DataWriter</code> matched with a multicast <code>DataReader</code>	144
	<code>DomainParticipant</code> was not rediscovered if it restarted before expiring	144
	Unexpected behavior when copying a <code>DDS_UnsignedShortSeq</code> with 0 length	144
	Local variables in header file may have caused compiler warning	145
	Non-default timer resolutions may have caused an incorrect timeout	145
	Missing checks for <i>max_routes_per_reader</i> and <i>max_routes_per_writer</i>	145
	Missing NULL checks for <i>enabled_transports</i>	145
	Possible exception due to misaligned RTPS header	145
	<code>DDS_SubscriptionBuiltinTopicData_copy</code> did not copy the <code>Presentation-QosPolicy</code>	146
	Possible failure to start timer	146
	Sample timestamp now set to 0 if timestamp cannot be retrieved	146
	<i>Qos_copy</i> functions did not validate input arguments	146
	Unused parameter <code>DOMAIN_PARTICIPANT_RESOURCE_LIMITS.matching_reader_writer_pair_allocation</code> removed	146
	<code>DDS_DomainParticipant_add_peer</code> may have returned success on failure	147
	<code>DDS_StringSeq_copy</code> did not validate input arguments	147
	Possible NULL pointer exception in generated code if the system was out of memory	147
	A <code>DataWriter</code> could run out of resources if sample was not added to cache	147
	Possible serialization beyond stream buffer	147
	<code>RELIABILITY.max_blocking_time</code> must be zero	148
	Possible <code>DataReader</code> or <code>DataWriter</code> creation failure with multiple <code>DomainParticipants</code>	148
	Incorrect <i>lease_duration</i> may have been used for a discovered participant	148
	Missing consistency check for <code>DESTINATION_ORDER.source_timestamp_tolerance</code>	148
	Improved error detection for unresolved addresses	148

<i>DDS_StatusCondition_set_enabled_statuses</i> did not trigger if an active condition was enabled	149
Race condition in DDS <i>enable</i> APIs	149
<i>disable_auto_interface_config</i> is not available in <i>Connex Cert</i>	149
DDS WaitSet may have timed out later than timeout value	149
10.5 Known Issues	149
10.5.1 Network interfaces cannot have non-ASCII characters in their name	149
10.5.2 Script errors when running from a directory with a space in its name	150
10.5.3 Low resolution timestamps may cause Zero Copy samples to remain visible longer than expected	150
10.5.4 DDS_WstringSeq instance may have inconsistent state when <i>set_maximum</i> fails	150
10.5.5 Logging mechanism is not included by default	150
10.5.6 DDS_DomainParticipantFactory_get_instance() not memory safe on some architectures	151
10.5.7 Samples cannot be reclaimed if Zero Copy application does not return loan	151
10.5.8 VOLATILE Zero Copy DataReaders behave as TRANSIENT_LOCAL	151
10.5.9 Reliable DataWriters send Heartbeats to best-effort DataReaders	152
10.5.10 Static endpoint discovery requires unique object IDs across all remote endpoints	152
11 Copyrights	153
12 Third-Party and Open Source Software	155
12.1 Connex Cert Libraries	155
12.1.1 <i>crc32c.c</i>	155
12.1.2 MD5	156
12.2 RTI Code Generator (<i>rtiddsgen</i>)	157
12.2.1 ANTLR	157
12.2.2 Apache Commons Lang	158
12.2.3 Apache Log4j 2	158
12.2.4 Apache Velocity	158
12.2.5 Gson	158
12.3 Appendix	158
12.3.1 Apache License version 2.0, January 2004 (http://www.apache.org/licenses/)	158
13 Contact Support	163
14 Join the Community	164

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

Key benefits of *Connex 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 Connex Professional* and *RTI Connex Micro*

Chapter 1

Introduction

1.1 What is RTI Connex Cert?

RTI Connex 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. *Connex Cert* uses the publish-subscribe communications model to make data distribution efficient and robust. *Connex Cert* simplifies application development, deployment and maintenance and provides fast, predictable distribution of time-critical data over a variety of transport networks. With *Connex 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.

Connex 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 *Connex 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. *Connex 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 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

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

Connex 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.

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

1.3.1 DDS Wire Compatibility

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

Submessage	Supported	DDS Standard	Connex 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

Connex Cert does not support mutable Qos policies.

Submessage	Supported	DDS Standard	Connex 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 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 Connex Documentation

Throughout this document, we may suggest reading sections in other *RTI Connex* documents. These documents are in your *RTI Connex* installation directory under **rti-connex-dds-<version>/doc/manuals**. A quick way to find them is from *RTI Launcher's* Help panel, select “Browse Connex 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 Connex Core Libraries User's Manual](#). These sections teach basic DDS concepts applicable to all RTI middleware, including *RTI Connex Professional* and *RTI Connex Cert*. You can open the [RTI Connex Core Libraries User's Manual](#) from *RTI Launcher's* Help panel.

The [RTI Community](#) provides many resources for users of DDS and the RTI Connex 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 Connex Cert is one of several products in the *RTI Connex* family of products:

RTI Connex Cert is a subset of *RTI Connex Micro*. *Connex 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 Connex 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. *Connex 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 Connex 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 Connex 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>/
  |--bin/
  |--doc/
  |--include/
  |--resource/
  |--rtiddsgen/
  |--CMakeLists.txt
  |--ReadMe.html
  +--src/
    +--rti_me_psl.1.0
```

- The `bin/` directory contains scripts for *rtiddsgen*.
- 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_connex_dds_cert-<version>-target-<architecture>.rtipkg`) adds the following to the *Connex Cert* directory:

```
--rti_connex_dds-<version>/
|
+--rti_connex_dds_cert-<version>/
|
+--lib/
|   +--<arch>CERT
|   |   +---<arch libraries>
```

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

2.3.1 Library types

Connex 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).

Connex Cert consists mainly of two libraries: a Platform Independent Library (PIL) provided by RTI and a Platform Support Library (PSL) that you build from source code. The PIL is an `rti_me` library that includes most of the functionality for *Connex 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. Some features, such as *Zero Copy Transfer*, are supported by a separate PIL from the `rti_me` library, as detailed in Table 2.1.

This is illustrated below:

The main benefit of splitting *Connex Cert* functionality into multiple 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 *Connex 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.

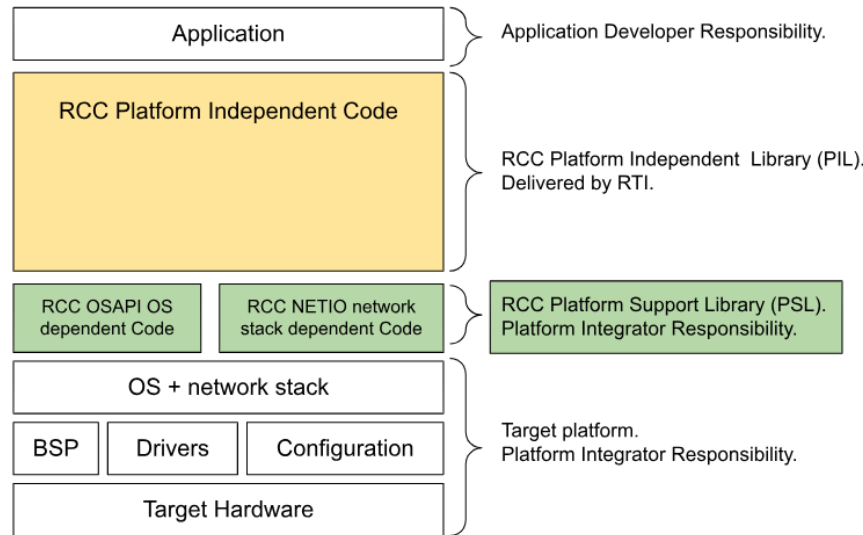


Figure 2.1: An overview of *Connex Cert* (RCC) libraries

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.

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
<code>rti_me</code>	Includes the following: • The core functionality for <i>Connext Cert</i>, including the DDS C API. • The Dynamic Participant Static Endpoint (DPSE) plugin. • The Reader and Writer History plugins.
<code>rti_me_netiozcopy</code>	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_cert-<version>
|  |--bin/
|  |--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 Connexrt Cert

Before you get started with *RTI Connexrt Cert*, you must build the Platform Support Library (PSL) for your platform from the source code provided in your *Connexrt 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: *Connexrt Cert* also requires a Platform Independent Library (PIL), which is provided as a pre-compiled binary in your *Connexrt 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.

- *Setting up the build environment*
 - *The host environment*
 - *The target environment*
- *Building the PSL*
 - *Building the PSL with rtime-make*
 - *Building the PSL with CMake*
 - * *Preparing to build*
 - * *Creating build files from the command line*
 - * *CMake flags used by Connexrt Cert*
- *Compile-time options*
 - *Debug information*
 - *Platform selection*

- *Compiler selection*
- *UDP options*
- *Custom build environments*
 - *Importing the Connex Cert code*

3.1 Setting up the build environment

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

- The *host* is the machine that runs the software to compile and link *Connex Cert*.
- The *target* is the machine that runs *Connex Cert*.
- In many cases *Connex 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 *Connex 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 *Connex Cert* is built, used, or run.

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

3.1.1 The host environment

Connex 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). *Connex Cert* does not require a case-sensitive file-system.
- have the necessary compiler, linkers, and build-tools installed.

The host machine **should**:

- have [CMake](http://www.cmake.org) (www.cmake.org) installed. Note that it is not required to use [CMake](http://www.cmake.org) to build *Connex Cert*, and in some cases it may also not be recommended. As a rule of thumb, if *Connex Cert* can be built from the command-line, [CMake](http://www.cmake.org) is recommended.
- be able to run bash shell scripts (Unix type systems) or BAT scripts (Windows machines).

Note: Linux systems are the only supported host environments for building *Connex Cert*. Typical examples of host machines are a Linux PC with the GNU tools installed (make, gcc, g++, etc.) or a Linux 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.

Connex 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, such as QNX or Linux.

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 *Connex Cert* because it simplifies configuring the *Connex 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 *Connex Cert*. This system makes it easier to support building *Connex 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 *Connex Cert* application. Refer to *Custom build environments* for more information on this option.

3.2.1 Building the PSL with rtime-make

The *Connex 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 *Connex 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 *Connex 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 *Connex 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.
- `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 Connex Cert

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

- `-DRTIME_TARGET_NAME=<name>` - The name of the target (equivalent 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 *Connex 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.

3.3 Compile-time options

The *Connex 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 *Connex Cert* debug information is independent of a debug build as defined by a compiler. In the context of *Connex 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 *Connex 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 *Connex 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 *Connex 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 Connex 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 *Connex Cert* and system libraries
 - Required compiler and linker flags
 - Details on how the *Connex 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 *Connex Cert* installation. For more information, see *Library types*.

4.1.2 Supported libraries by platform

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

Table 4.1: Supported Libraries by Platform

Platform	RTI Architecture	Supported Libraries
QOS 8.0 (QNX® OS for Safety)	armv8leElfqcc12.2.0CERT	rti_me rti_me_netiozcopy
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 8.0 (QNX® OS for Safety)	armv8leElfqcc12.2.0CERT	✓	✓ ¹	✓ ¹
QOS 2.2.1 (QNX OS for Safety)	armv8leElfqcc8.3.0CERT	✓	✓ ¹	✓ ¹

¹ This transport is not implemented in the PIL. The RTI-provided PSL includes a reference implementation of this transport.

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)	8.0.2	ARMv8 (64-bit)	OS Default: io-sock	qcc_gpp12.2.0	armv8leElfqcc12.2.0CERT	armv8leElfqcc12.2.0CERT-QOS8.0
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
armv8leElfqcc12.2.0CERT	Static Release	-Vgcc/12.2.0,gcc_ntoaarch64le -std=c99 -Winit-self -fno-strict-aliasing -Wmissing-declarations -Wall -Wextra -Wpedantic -Wshadow -Wcast-align -Wunused -Wconversion -Wsign-conversion -Wlogical-op -Wdouble-promotion -DNDEBUG -O2 -fpic
armv8leElfqcc8.3.0CERT	Static Release	-Vgcc/8.3.0,gcc_ntoaarch64le -std=c99 -Winit-self -fno-strict-aliasing -Wmissing-declarations -Wall -Wextra -Wpedantic -Wshadow -Wcast-align -Wunused -Wconversion -Wsign-conversion -Wlogical-op -Wdouble-promotion -DNDEBUG -O2 -fpic

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 Connex 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 *Connex 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 `rti-time-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
armv8leElfqcc12.2.0CERT-QOS8.0	Static Release	-Vgcc/12.2.0,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
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

Chapter 5

Getting Started

This section will help you get started with *Connexrt 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 *Connexrt Cert*'s features.

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

5.1 GitHub Example

Connexrt 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 *Connexrt Cert* does not include a C++ API.

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

5.2 Generating Examples

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

Note: Before running *rtiddsgen*, you might need to add `rti_connexrt_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 armv8leElfqcc8.3.0CERT-QOS2.2.1 --
↪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 armv8leElfqcc8.3.0CERT-QOS2.2.1`, then the example applications should be compiled with the `armv8leElfqcc8.3.0CERT-QOS2.2.1` 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/armv8leElfqcc8.3.OCERT-QOS2.2.1/<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/armv8leElfqcc8.3.OCERT-QOS2.2.1/<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 *Connexrt 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 *Connexrt 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 *Connexrt 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 Connexrt Cert*. If you installed *RTI Connexrt* with default settings, *RTI Connexrt Cert* will be here: `<path_to_connexrt_dds_installation>/rti_connexrt_dds-<version>/rti_connexrt_micro-<version>`. If you copied *RTI Connexrt Cert* to another place, set `RTIMEHOME` to point to that location.

6.1.2 Add required preprocessor flags

All *Connexrt 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 *Connex 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 *Connex 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 *Connex 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.

Connex 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 *Connex 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 *Setting up the Datagram UDP* 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 “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, "dpse"))
{
    /* 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("_udp://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](#) API for asserting a remote publication must be called for each remote *DataWriter* that a *DataReader* may discover.

```

struct DDS_PublicationBuiltinTopicData rem_publication_data =
    DDS_PublicationBuiltinTopicData_INITIALIZER;

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

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

rem_publication_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

retcode = DPSE_RemotePublication_assert(participant,
                                       "Participant_1",
                                       &rem_publication_data,
                                       HelloWorld_get_key_kind(HelloWorldTypePlugin_
↳get(),
                                       NULL)));

if (retcode != DDS_RETCODE_OK)
{
    /* failure */
}

```

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

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 Connex Cert Library

Connex Cert has been designed to integrate with a wide range of operating systems, network stacks, and CPUs. For this reason, *Connex Cert* places few restrictions on how it is integrated. The memory management API defined by *Connex Cert* may be implemented using standard C library 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, *Connex 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.

Connex 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 guaranteed 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 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 Connex 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 *Connex 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 *Connex Cert*, you must define that type in a way the middleware understands and then register the type with the middleware. These steps allow *Connex 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 Connex 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 *explicit* 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

Connex 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 *Connex 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 *explicit* 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 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 *Connex 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 (QoS Policies). 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 Connex Cert* does not support the following:

- **MultiTopic**
- **ContentFilteredTopic**
- **ReadCondition**
- **QueryConditions**

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

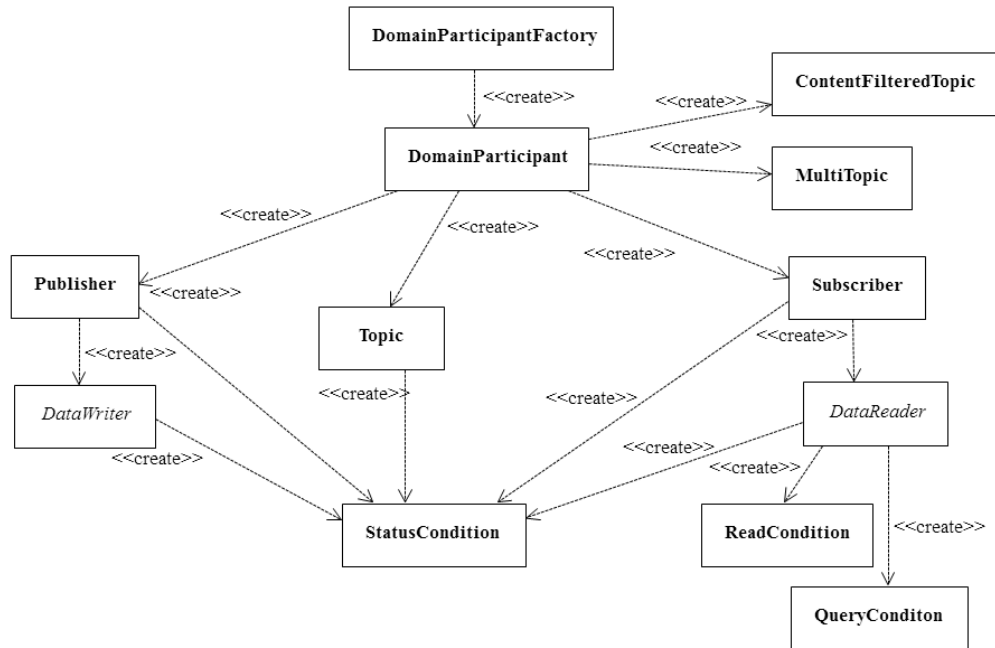


Figure 7.1: Overview of DDS Entities

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* internal buffers from where the DDS data sample is sent under the criteria set by the *Publisher*’s and *DataWriter*’s QoS Policies. 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 QoS Policies. A *Listener* is how *Connext* notifies your application of status changes relevant to the *Entity*. The QoS Policies 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 QoS Policies 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 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.

- 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*.

- 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:

- 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.'
- Create a *WaitSet*.
- Attach the *StatusCondition* to the *WaitSet*.
- 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).
- 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 QoS Policies. The *Listener* is how *RTI Connext Cert* notifies your application of status changes relevant to the *Entity*. The QoS Policies 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 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 QoS Policies.

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.

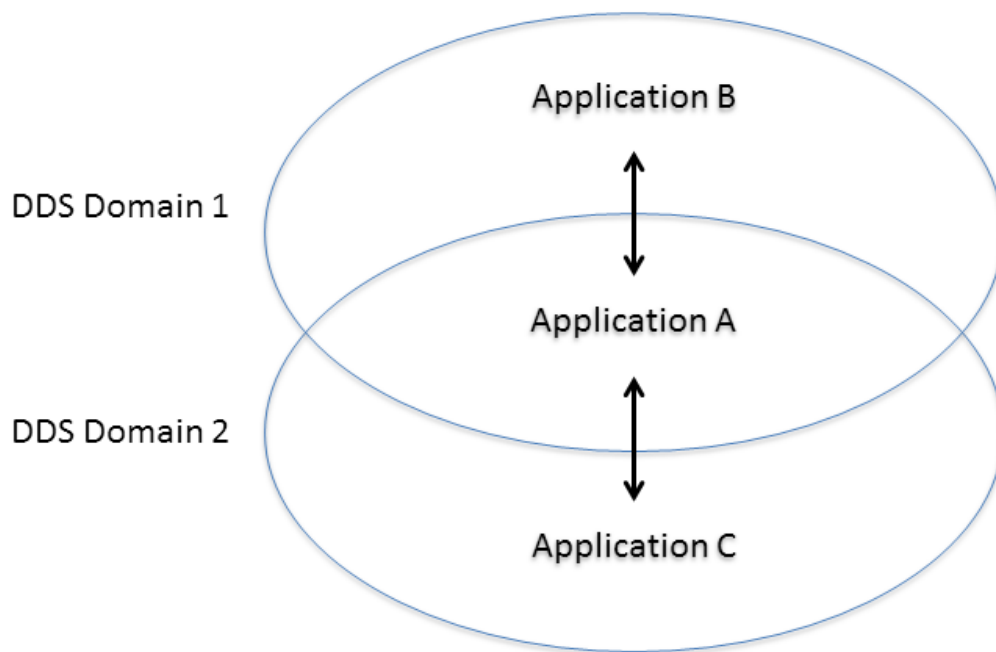


Figure 7.2: Relationship between Applications and DDS Domains

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 QoS Policies and *Listeners*. The *DomainParticipant* entity also allows you to set ‘default’ values for the QoS Policies for all the entities created from it or from the entities that it creates (*Publishers*, *Subscribers*, *Topics*, *DataWriters*, and *DataReaders*).

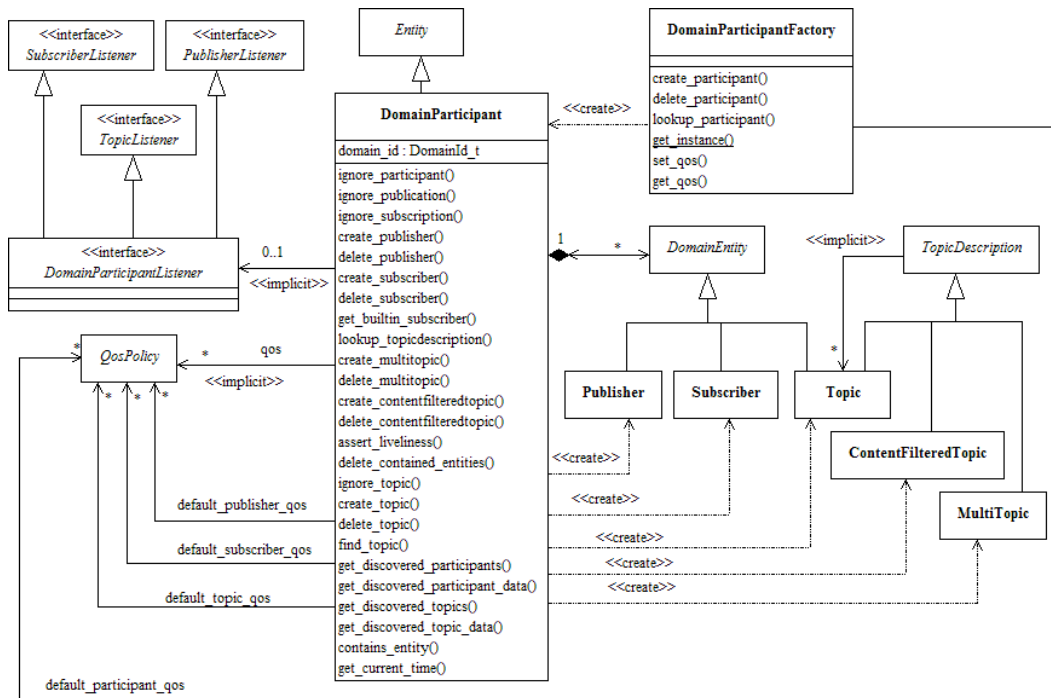


Figure 7.3: DDS Domain Module
Note: MultiTopics are not supported.

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 exchange 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 registrations 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 entities 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* identifies 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.

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 *Connex 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_]+
```

(continues on next page)

(continued from previous page)

```

INTEGER = DEC_INTEGER | HEX_INTEGER

DEC_INTEGER = [0-9]+

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

ADDRESS = 0 or more 8bit characters

```

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;

    /* 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;
    }
}

```

(continues on next page)

(continued from previous page)

```

    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`: *Connex 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;
    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* endianness.
2. The number returned in `checksum64` is assumed to be in *host order* endianness.
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,
```

(continues on next page)

(continued from previous page)

```

                                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;

    /* 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");
    }
}

```

(continues on next page)

(continued from previous page)

```

        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;
    }

    /* 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. */
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_Interface_Factory_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 *Connex 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_udp4,
    MyDgramInterface_send,
    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);
*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 integration with a UDP transport. This usually happens via invocation of a UDP transport convenience function like `UDpv4_Interface_register()` with UDP properties to create the datagram instance for UDP, as described in the following table:

Table 7.1: UDP Transport Functions and Properties

Name	Description
<code>UDpv4_Interface_register</code>	Function that registers the UDP interface.
<code>UDpv4_TransportProperty_set_max_message_size</code>	Property that sets the maximum message size for the UDP transport.
<code>UDpv4_TransportProperty_set_max_receive_buffer_size</code>	Property that sets the maximum size of the UDP transport buffer for receiving messages.
<code>UDpv4_TransportProperty_set_max_send_buffer_size</code>	Property that sets the maximum size of the UDP transport buffer for sending messages.

Warning: The `UDpv4_Interface_register` function and UDP properties are included in the reference Platform Support Library (PSL) provided by RTI, *not* the *Connex Cert* Platform Independent Library (PIL). This means they are not part of the certifiable code in *Connex Cert*. A different UDP transport implementation (as provided by your own PSL) may require different functions and properties.

Refer to the following code snippet for an example:

```
UDpv4_TransportProperty_T *udp_property = UDpv4_TransportProperty_new();

/* Add a network interface to the UDpv4 transport.
 * 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,
    0xff000000,
    "lo",
    UDP_INTERFACE_INTERFACE_UP_FLAG |
    UDP_INTERFACE_INTERFACE_MULTICAST_FLAG))
{
    /* error */
}
```

(continues on next page)

(continued from previous page)

```

/* Set the UDPv4 transport's buffer properties. Error checking is omitted for
 * the sake of brevity. */
UDPV4_TransportProperty_set_max_send_buffer_size(udp_property, MAX_SEND_BUFFER_SIZE);
UDPV4_TransportProperty_set_max_receive_buffer_size(udp_property, MAX_RECV_BUFFER_SIZE);
UDPV4_TransportProperty_set_max_message_size(udp_property, MAX_MESSAGE_SIZE);

/* Call the UDPv4 transport's convenience function to register it via the
 * Datagram transport. */
if (!UDPV4_Interface_register(registry, "_udp", udp_property))
{
    /* error */
}

```

Enabled transports can be configured with "_udp://". This will use all interfaces. Enabling 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 name that the transport was registered with */
*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 7.2: Structure for the User Interface

Interface Attribute	Description
<code>create_instance</code>	Creates an instance of the <code>NETIO_DGRAM</code> interface.
<code>delete_instance</code>	Deletes an instance of the <code>NETIO_DGRAM</code> interface.
<code>get_interface_list</code>	Reads the available interfaces from the <code>NETIO_DGRAM</code> interface.
<code>release_address</code>	Instructs the <code>NETIO_DGRAM</code> interface to stop listening for messages on the source address.
<code>resolve_address</code>	Instructs the <code>NETIO_DGRAM</code> interface to determine if the address string is valid.
<code>send</code>	Instructs the <code>NETIO_DGRAM</code> interface to send a message.
<code>get_route_table</code>	Instructs the <code>NETIO_DGRAM</code> interface <code>netio_intf</code> to return a sequence of address and netmask pairs that this interface can send to.
<code>bind_address</code>	Instructs the <code>NETIO_DGRAM</code> 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 processed 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. *Connex 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 7.3: Default User-Defined Properties for Zero Copy v2

Property Name	Description	Default
<code>user_property.intf_addr</code>	Sets the interface address for this instance. Should be unique for all instances in the system.	0
<code>user_property.thread_prop.stack_size</code>	Sets the stack size for the receive thread created for receiving notifications.	OSAPI_THREAD_PROPERTY_DEFAULT
<code>user_property.thread_prop.priority</code>	Sets the priority for the receive thread created for receiving notifications.	OSAPI_THREAD_PROPERTY_DEFAULT
<code>user_property.thread_prop.options</code>	Sets any thread options supported by the thread API implementation.	OSAPI_THREAD_PROPERTY_DEFAULT
<code>user_property.max_receive_ports</code>	Sets the maximum number of receive ports that can be opened. Used as part of the <code>user_intf.bind</code> function.	2
<code>user_property.max_routes</code>	Sets the maximum number of outgoing routes. Used as part of the <code>user_intf.add_route</code> 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 7.4: User Interface Properties for Zero Copy v2

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

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 Connex 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 QoS Policies that are part of the *DomainParticipant's* built-in data need to be propagated.

When receiving remote participant discovery information, *Connex 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 *Connex Cert* (you may be able to change it in a future version).

7.8.2 Configuring Participant Discovery Peers

An *RTI Connex 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.

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 announcements to multicast address 239.255.0.1
 * using the transport registered as _udp. Note that for multicast
 * addresses the index is not relevant since it is a shared address.
```

(continues on next page)

(continued from previous page)

```

*/
*DDS_StringSeq_get_reference(&dp_qos.discovery.initial_peers, 1) =
    DDS_String_dup("_udp://239.255.0.1");

/* Send announcements 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 announcements 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 announcements 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 announcements. 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>

```

(continues on next page)

(continued from previous page)

```
PREFIX = [a-zA-Z_][0-9a-zA-Z_]+
ADDRESS = 0 or more 8bit characters
```

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 announcements. 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
```

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");

*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 only provides one way of determining endpoint information of other *DomainParticipants* *Static Discovery Plugin*.

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 acronym DPSE to refer to the static endpoint discovery plugin in *RTI Connext Cert*. The DPSE plugin implements the SPDP 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. In [DPSE](#), the remote endpoint's QoS must be configured locally. With remote endpoints asserted, the *DomainParticipant* then waits until it receives a participant discovery announcement from an asserted remote *DomainParticipant*. Once received that, all endpoints that have been asserted for that remote *DomainParticipant* are considered discovered and ready to be matched with local endpoints.

Assume participant_1 contains a *DataWriter*, and participant_2 has a *DataReader*, both communicating on topic HelloWorld. participant_1 needs to assert the *DataReader* in participant_2 as

a remote subscription. The remote subscription data passed to the operation must match exactly the QoS actually used by the remote *DataReader*:

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

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

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

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

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

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

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

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.

7.8.9 DomainParticipant Discovery by Name

If a *DomainParticipant* is restarted after an ungraceful shutdown, other *DomainParticipants* will not remove the *DomainParticipant*'s discovery information until its `lease_duration` has expired. Typically, you can reduce the `lease_duration` or increase the resource limits to avoid running out of resources to discover restarted *DomainParticipants*. However, this may not be a feasible solution, since it increases network load and memory usage.

To mitigate these issues, *Connext Cert* can be configured to discover *DomainParticipants* by name instead of by GUID. This speeds up the discovery process because the restarted *DomainParticipant* can immediately replace the old *DomainParticipant* without waiting for the old one's `lease_duration` to expire.

To enable discovery by name for a *DomainParticipant*, do the following:

1. Set a globally unique `participant_name` in the *DomainParticipant*'s [DomainParticipantQos](#).
2. Set the `enable_participant_discovery_by_name` property to `DDS_BOOLEAN_TRUE` in the *DomainParticipant*'s [DISCOVERY](#) QosPolicy.

The *DomainParticipant* can now be rediscovered smoothly by other remote *DomainParticipants* after a restart.

Attention: When using discovery by name, a *DomainParticipant* must be restarted with a different GUID from its previous GUID.

Attention: To avoid undefined behavior while using discovery by name, make sure that each *DomainParticipant* in a given domain has a unique `participant_name`.

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 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-<version>`
- `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_connex_micro_root/rtiddsgen/scripts/rtiddsgen -micro -language C -replace
UserType.idl
```

```
> $rti_connex_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 *Connex 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 *Connex 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/initialization and deletion (only for *Connex 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 *Connex 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 *Connex 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

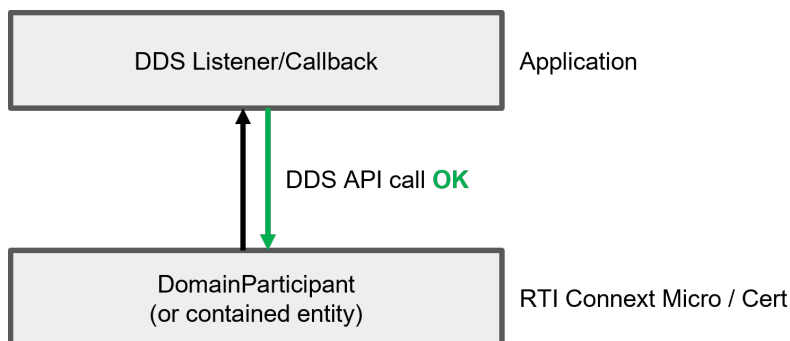
All public APIs have a note about thread-safety included in the API reference manuals. It is important that an application does not violate thread-safety guidelines. *RTI Connext Cert* creates one critical section per *DomainParticipant* to protect resources used by the *DomainParticipant*.

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.

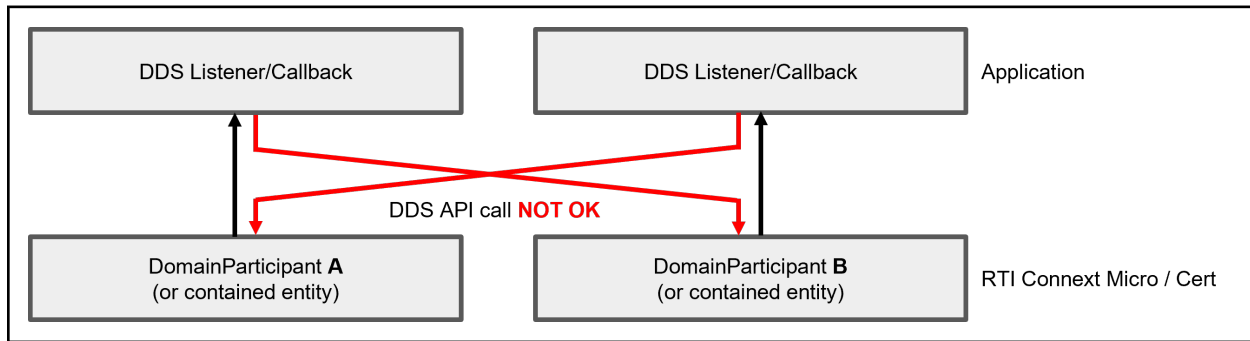
7.10.1 Calling DDS APIs from listeners and callbacks

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\(\)](#).

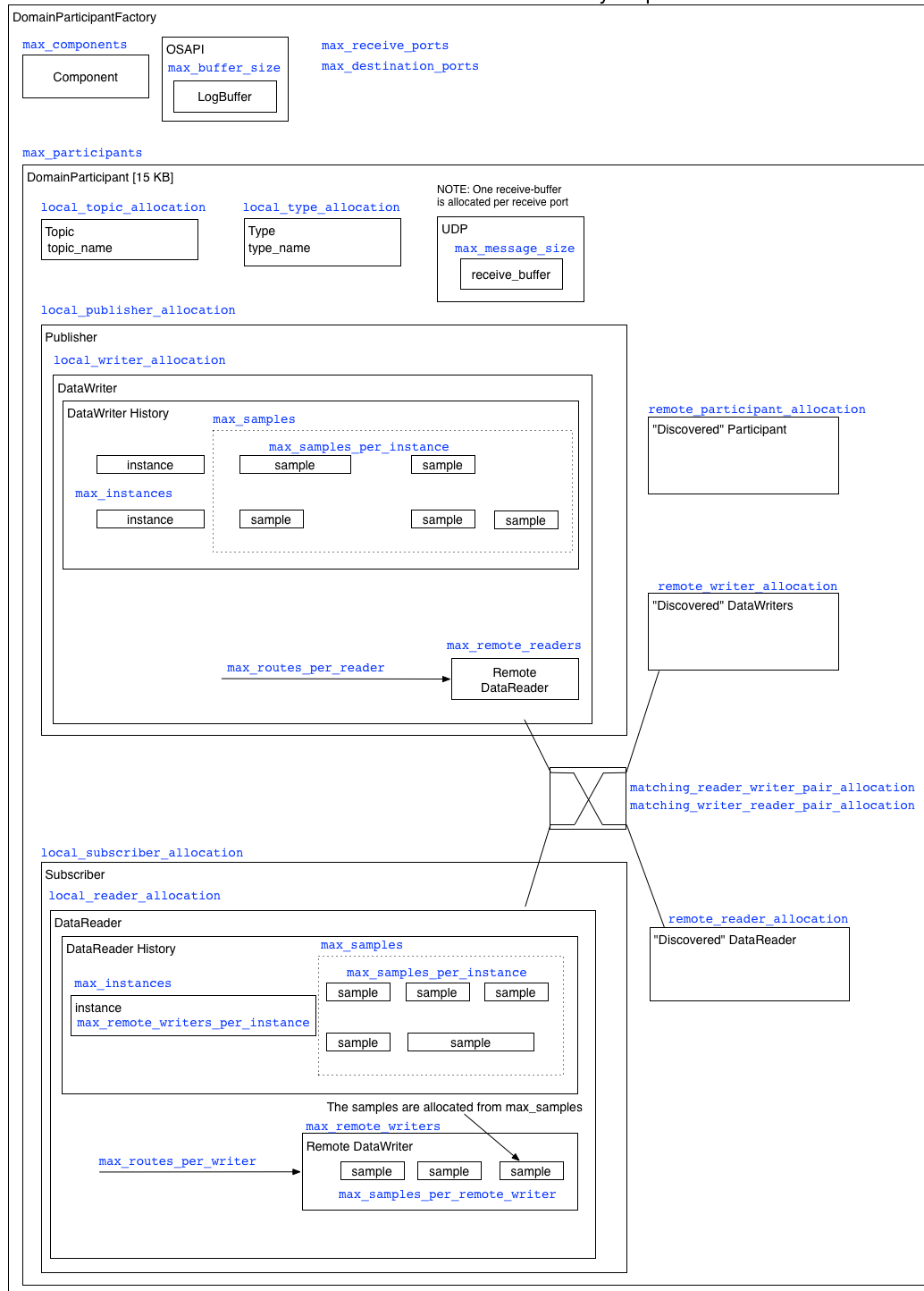
The *Connext Cert* user is supposed to use API function which allocate memory only during system initialization. This avoid to run into situations where a resource can not be allocated after the initialization phase.

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 Connex Micro Memory Map



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.

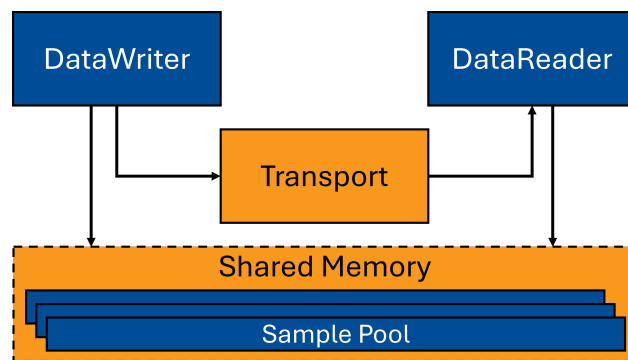


Figure 7.4: Zero Copy Transfer in *Connext Cert*

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.
- Enabling the Zero Copy transport causes *Connext Cert* to create one extra timer object. This means that each *DomainParticipant* using Zero Copy consumes 2 timers. Depending on the platform that provides the timers, as implemented in the Platform Specific Library (PSL), this may limit the number of *DomainParticipants* that can be created in a single process. For example, a PSL implementation that has an upper limit of instantiating 8 timers results in a maximum of 4 *DomainParticipants* per process.

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.

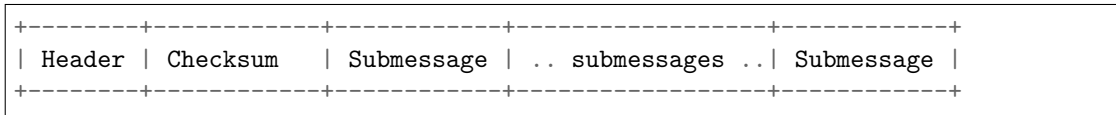
For information on how to write custom checksum functions, please refer to *RTPS*.

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:



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



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

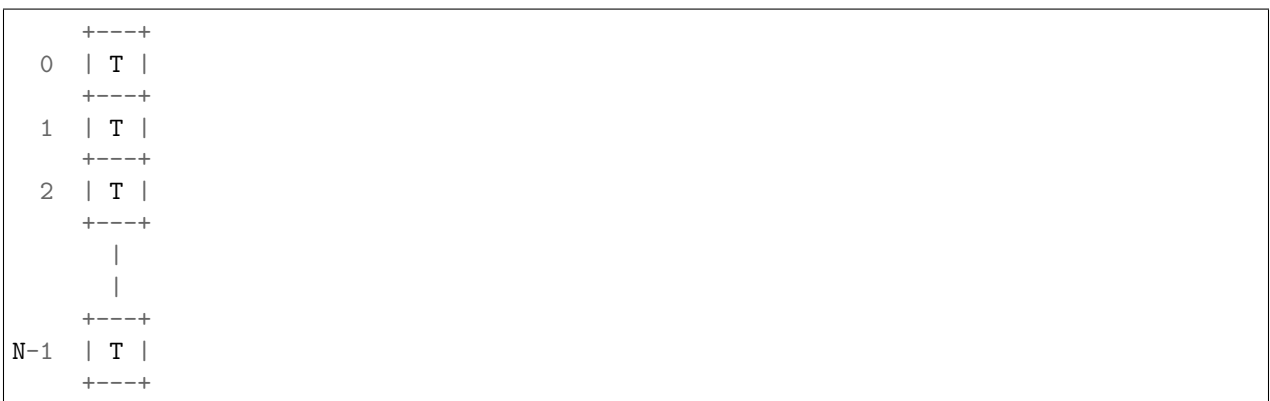
7.14.1 Introduction

RTI Connex 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.



There are three types of sequences in *Connex 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	Connex Cert Type	Connex 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 *Connex 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 once using **set__maximum()**.
- 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 *Connex 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;
    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_SEQUENCE_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_SEQUENCE_INITIALIZER;

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

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

Working with Application Defined Sequences

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

- The maximum size is 0 by default. It is necessary to call `set_maximum` 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. Use `DDS_StringSeq_set_maximum_w_max` to also allocate pointers.
- Note that in *Connext Cert*, `__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_set_maximum(&my_seq, 20);
DDS_StringSeq_set_length(&my_seq, 10);

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.

If logging is enabled, an error message is logged when a resource limit is exceeded. 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.

7.15.2 Summary

Resource	Limit
Number of domain participants per OS process	16
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.

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: Installing additional discovery plugins is not supported.
- 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.

7.16 Debugging

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

Connext Cert maintains a log of events occurring 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_INIIALIZER;

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)
```

(continues on next page)

(continued from previous page)

```
{
    write(1, buffer, length);
}
```

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

```
struct OSAPI_LogProperty prop = OSAPI_LogProperty_INITIALIZER;

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 *Connex 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 *Connex Cert*.

You can also configure *Connex 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 Connex Cert and RTI Connex

In some cases, it may be necessary to write an application that is compiled against both *RTI Connex Micro*, *RTI Connex Cert*, and *RTI Connex*. In general this is not easy to do because *RTI Connex Micro* and *RTI Connex Cert* supports a very limited set of features compared to *RTI Connex*. However, while *RTI Connex Cert* is a subset of *RTI Connex 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 Connex* 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 Connex Cert*. Note that *RTI Connex* 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 Connex Cert* and *RTI Connex* with respect to library names, header files, etc. It is advisable to keep the two installations separate, which is the normal case.

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

RTI Connex uses the environment variable `NDDSHOME` to locate the root of the *RTI Connex* 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 static discovery (DPSE), but you must 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_DomainParticipantResourceLimitsQosPolicy::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.ConnnextMicroCompatibility`.

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 Connex 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 *Connex Cert*), Replay is compatible with *RTI Connex 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 Connex Micro](#) for more information on how to convert from IDL to XML.

RTI Converter

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

8.6.8 Wireshark

Wireshark fully supports *RTI Connex Cert*.

8.6.9 Persistence Service

RTI Connex 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. Follow the link below in order to access the corresponding API Reference HTML documentation.

Note: *Connext Cert* 2.4.16.1 uses the 2.4.16 API Reference documentation because there are no API changes in 2.4.16.1.

- [C API Reference](#)

Chapter 10

Release Notes

10.1 Supported Platforms and Programming Languages

Connexrt 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
Ubuntu 18.04 LTS ¹	x64	gcc 7.3.0	x86_64leElfgcc7.3.0CERT
QOS 8.0 (QNX® OS for Safety)	ARMv8	qcc_gpp12.2.0	armv8leElfqcc12.2.0CERT
QOS 2.2.1 (QNX® OS for Safety)	ARMv8	qcc_gpp8.3.0	armv8leElfqcc8.3.0CERT

10.2 What's New in 2.4.16.1

Connexrt Cert 2.4.16.1 is a General Access Release (GAR) based on *Connexrt Cert* 2.4.16.

The following features are new since *Connexrt Cert* 2.4.16.

¹ Binaries for this platform are provided for development and testing purposes only. No certification evidence is available.

10.2.1 Improve system robustness with DomainParticipant rediscovery

This release adds the ability for a *DomainParticipant* with its `DomainParticipantQos.participant_name.name` set to a non-empty value to be rediscovered if it is restarted before its `lease_duration` expires. You can enable this feature by setting the `DomainParticipantQos.discovery.enable_participant_discovery_by_name` property to `DDS_BOOLEAN_TRUE`.

For more information on this feature, refer to the *DomainParticipant Discovery by Name* section.

10.2.2 Added support for QOS 8.0

This release adds support for the following new platform:

OS	CPU	Compiler	RTI Architecture Abbreviation
QOS 8.0 (QNX® OS for Safety)	ARMv8	qcc_gpp12.2.0	armv8leElfqcc12.2.0CERT

Refer to *QNX Platforms* for more information on how to build *Connext Cert* for this platform.

10.3 What's Fixed in 2.4.16.1

The following are fixes since *Connext Cert* 2.4.16.

[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] When using Zero Copy, the sample returned by `get_loan()` could have an insufficient size

When certain data types were enabled for the Zero Copy v2 transport, the value returned by `sizeof()` was larger than the difference between the pointers returned by the *DataWriter*'s `get_loan()` method. As a consequence, applications were at risk of writing outside of allocated memory boundaries.

This problem has been resolved by updating an `rtiddsgen` code template. You must regenerate any code previously generated by `rtiddsgen` after installing the modified template.

[RTI Issue ID MICRO-12511]

10.3.2 [Major] Incorrect check for multicast address

Note: This issue only affected the reference PSL, not the PIL.

To determine if an IP address was a multicast address, *Connex Cert* should have checked if the first byte of the address fell within the range of 224-239. However, *Connex Cert* only checked if the address started with 239. If an application did not use 239 exactly, then the application would not join the multicast group and *DomainParticipant* creation could fail.

[RTI Issue ID MICRO-12795]

10.3.3 [Major] `udpv4_singlethr_init_port_entry()` function caused UDP communication to fail

Note: This issue only affected the reference PSL, not the PIL.

The function `udpv4_singlethr_init_port_entry()` did not initialize all fields of the `struct sockaddr_in sock_addr` object. This could cause the binding of a socket to fail, which then caused communication over the UDP transport to fail.

[RTI Issue ID MICRO-12765]

10.4 Previous Releases

10.4.1 What's New in 2.4.16

The following features are new since *Connex Cert* 2.4.15.

Deprecations and Removals

This section describes items that are deprecated or removed in *Connex Cert* 2.4.16, compared to *Connex Cert* 2.4.15.

Deprecated means that the item is still supported in this release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported.

Any deprecations or removals noted in RTI's documentation serve as notice under the Real-Time Innovations, Inc., Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software. RTI's current standard terms and support and maintenance policies are available at <https://www.rti.com/terms>.

Removed support for specifying locator addresses without a transport prefix

Connex Cert no longer supports the ability to specify a locator address (such as 127.0.0.1) without a transport name prefix. For example, if you register a transport with the name `udp` and for the address 127.0.0.1, you must specify the locator as `udp://127.0.0.1`. Using a locator address without a transport name prefix will cause the associated *DomainParticipant* or endpoint creation to fail and return NULL.

10.4.2 What's Fixed in 2.4.16

The following are fixes since *Connex Cert* 2.4.15.

[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.

Discovery

[Critical] Lost INFO_DST messages with GUIDPREFIX_UNKNOWN

Connex Cert may have silently discarded RTPS INFO_DST messages containing a *guidPrefix* of 0x0 (GUIDPREFIX_UNKNOWN).

[RTI Issue ID MICRO-8751]

[Major] DomainParticipants continued to send DATA(p) messages to nonexistent remote participants

A *DomainParticipant* might have continued to send DATA(p) discovery messages to remote *DomainParticipants* that were once discovered but no longer exist.

[RTI Issue ID MICRO-8747]

Reliability Protocol and Wire Representation

[Major] A reliable `DataReader` may have stopped receiving data

A reliable *DataReader* may have stopped receiving data if the data was received out of order from a *DataWriter* and the *DataWriter* was deleted before the *DataReader* could request the missing data.

[RTI Issue ID MICRO-10317]

Generated Code (C)

[Major] Code Generator generated examples using IP addresses without transport prefix

Examples generated by *rtiddsgen* were missing a required transport prefix (`_udp://`).

[RTI Issue ID MICRO-10519]

APIs (C)

[Major] `DDS_Publisher_lookup_datawriter` returned a `DataWriter` created by a different Publisher

The `DDS_Publisher_lookup_datawriter` API searches for a *DataWriter* for a given *Topic* created by the *Publisher*. However, the returned *DataWriter* could belong to a different *Publisher* if multiple *DataWriters* were created for the same *Topic* in different *Publishers*. Now, `DDS_Publisher_lookup_datawriter` will only return *DataWriters* that were created by the original *Publisher*.

[RTI Issue ID MICRO-4570]

[Major] `DDS_Subscriber_lookup_datareader` returned a `DataReader` created by a different Subscriber

The `DDS_Subscriber_lookup_datareader` API searches for a *DataReader* for a given *Topic* created by the *Subscriber*. However, the returned *DataReader* could belong to a different *Subscriber* if multiple *DataReaders* were created for the same *Topic* in different *Subscribers*. Now, `DDS_Subscriber_lookup_datareader` will only return *DataReaders* that were created by the original *Subscriber*.

[RTI Issue ID MICRO-4569]

[Minor] Incorrect error code for DDS_DomainParticipantQos_initialize and DDS_DomainParticipantQos_copy

When an invalid parameter was used to invoke `DDS_DomainParticipantQos_initialize` or `DDS_DomainParticipantQos_copy`, *Connex Cert* would incorrectly return a *PRECONDITION_NOT_MET* error. *Connex Cert* now returns the correct error, *BAD_PARAMETER*.

[RTI Issue ID MICRO-9635]

Memory Leaks/Growth**[Major] Resources were not freed when calling unregister_instance on a reliable Zero Copy DataWriter**

When using a reliable Zero Copy *DataWriter*, calling `FooDataWriter_unregister_instance()` would not return the instance's resources to the appropriate pool for reuse.

[RTI Issue ID MICRO-10039]

Data Corruption**[Critical] Historical samples overwritten in the DataWriter queue when using TRANSIENT_LOCAL over UDPv4**

When applications communicated using `TRANSIENT_LOCAL` over the UDPv4 transport, if a *DataWriter* wrote any samples before establishing communication with a *DataReader*, the samples would be overwritten in the *DataWriter* queue when they were sent as historical data.

[RTI Issue ID MICRO-10469]

Interoperability**[Major] Zero Copy communication failed when applications were started from different directories**

When two applications were started from different working directories, they would fail to communicate when using the Zero Copy v2 transport.

Note: As a result of this fix, the Zero Copy transfer feature is not backwards compatible. Applications using an older version of *Connex Cert* Zero Copy transfer will not be able to communicate

with applications using Zero Copy transfer from 2.4.16 onward.

[RTI Issue ID MICRO-10391]

[Minor] OSAPI_SharedMemorySegment_get_max_size() functioned incorrectly on QNX systems

In the previous release, Linux and QNX systems shared the same implementation of the `OSAPI_SharedMemorySegment_get_max_size()` function. This function only supported the `/proc/sys/kernel/shmmax` path on Linux systems; calling `fopen()` on this path on a QNX system would fail and return the default shared memory maximum size.

This release continues to use `/proc/sys/kernel/shmmax` for Linux systems, but now returns the default shared memory maximum size directly for QNX systems without requiring a failed call to `fopen()`.

Note: This fix was made to the reference Platform Support Library (PSL) implementation, which is included in the PSL source as an example. The reference PSL is *NOT* part of certified product code.

[RTI Issue ID MICRO-8879]

Other

[Major] Possible race condition with Zero Copy shared memory API

A race condition could occur if you accessed the Zero Copy shared memory API before initialization had been completed.

[RTI Issue ID MICRO-10019]

[Major] DataWriter with finite timestamp failed to send data if date exceeded January 2038

A *DataWriter* configured with a finite source timestamp tolerance and `DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS` would fail to send data after 03:14:07 UTC 19 January 2038.

[RTI Issue ID MICRO-5858]

[Major] Connex Cert would not work if year exceeded 2038

If the date was set beyond the year 2038, *Connex Cert* would not work. This was because the date was reported as a 32-bit unsigned integer; however, *Connex Cert* expected a signed 32-bit value and would therefore interpret the “wrap around” value as a negative number, causing an error.

[RTI Issue ID MICRO-2295]

[Minor] Multiple endpoints did not match properly when using Zero Copy transfer

When using the Zero Copy v2 transport, multiple endpoints would not match properly if both *DomainParticipants* were enabled before creating *DataReaders* or *DataWriters*.

[RTI Issue ID MICRO-10162]

[Minor] SYSTEM_RESOURCE_LIMITS.max_components QoS setting documented incorrectly

Connex Cert documentation incorrectly stated that the `SYSTEM_RESOURCE_LIMITS.max_components` QoS setting could be changed. This has been corrected to state that it cannot be changed. The default value has also been increased to 64.

[RTI Issue ID MICRO-3571]

[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]

10.4.3 What's New in 2.4.15

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

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

This release splits *Connex Cert* into two libraries:

- Platform Independent Libraries (PIL): binaries that support the basic features of *Connex 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 *Connex Cert* code. Previous releases of *Connex 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.

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*.

10.4.4 What's Fixed in 2.4.15

The following are fixes since *Connex 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.

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

Connex 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]

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

Previously, *Connex 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.

Connex Cert and *Connex 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]

[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: *Connex Cert* does not support the `@default` annotation.

[RTI Issue ID MICRO-6402]

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

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

[RTI Issue ID MICRO-5148]

[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]

[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]

[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]

[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]

[Major] Possible segmentation fault when a Zero Copy-enabled DataReader called read() after unmatched 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]

[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]

[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]

[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]

[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]

[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]

10.4.5 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 *Connex 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 *Connex Cert*. The default send and receive socket buffer size has been changed to 64 Kbytes in *Connex Cert* for QNX builds.

10.4.6 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 architectures 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_DomainParticipant_add_peer* may have returned success on failure**

DDS_DomainParticipant_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 previous 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 *Connex Cert

In previous versions of *Connex Cert*, *disable_auto_interface_config* was available in the *UDP_InterfaceFactoryProperty* structure, but was ignored (always TRUE). This variable is no longer available in *Connex 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 Network interfaces cannot have non-ASCII characters in their name

If a network interface name contains one or more non-ASCII characters, creating associated *DomainParticipants* may fail and return NULL.

As a workaround, do not use non-ASCII characters in network interface names.

[RTI Issue ID MICRO-11127]

10.5.2 Script errors when running from a directory with a space in its name

If the path where the `rtime-make` script resides includes a space, invoking `rtime-make` will cause a scripting error that includes the following: `line 25: cd: too many arguments`.

As a workaround, install *Connex Cert* in a directory with no spaces or other special characters in its path.

[RTI Issue ID MICRO-11099]

10.5.3 Low resolution timestamps may cause Zero Copy samples to remain visible longer than expected

While using the Zero Copy v2 transport, if a sample is pushed out of a *DataWriter*'s history at the same time (with the same timestamp) as a *DataReader* is created, the *DataReader* will still be able to observe it. Additionally, Zero Copy samples with timestamps using a different clock than the *DataReader* may be missed.

There is no known workaround for this issue.

[RTI Issue ID MICRO-10926]

10.5.4 DDS_WstringSeq instance may have inconsistent state when set_maximum fails

If the function `DDS_WstringSeq_set_maximum()` fails and returns `RTI_FALSE`, the entire sequence object may have an inconsistent state.

As a workaround, do not use the `DDS_WstringSeq` instance after a call to its `set_maximum` has failed.

[RTI Issue ID MICRO-9999]

10.5.5 Logging mechanism is not included by default

Connex 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]

10.5.6 DDS_DomainParticipantFactory_get_instance() not memory safe on some architectures

On architectures with weak memory ordering (such as ARM), `DDS_DomainParticipantFactory_get_instance()` is not guaranteed to be memory safe.

As a workaround, use an OS API call that acts as a memory barrier before and after calling `DDS_DomainParticipantFactory_get_instance()`. For example:

```
mutex_lock(my_mutex);
instance = DDS_DomainParticipantFactory_get_instance();
mutex_unlock(my_mutex);
```

[RTI Issue ID MICRO-8004]

10.5.7 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]

10.5.8 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]

10.5.9 Reliable DataWriters send Heartbeats to best-effort DataReaders

Reliable *DataWriters* incorrectly send periodic and piggybacked Heartbeats to best-effort *DataReaders*.

[RTI Issue ID MICRO-2225]

10.5.10 Static endpoint discovery requires unique object IDs across all remote endpoints

When using static endpoint discovery (DPSE), *Connex 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

© 2019-2026 Real-Time Innovations, Inc.

All rights reserved.

Printed in U.S.A. First printing.

September 2026.

Trademarks

Real-Time Innovations, RTI, NDDS, Connex, the RTI logo, 1RTI and the phrase, “Your Systems. Working as one,” are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

This is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

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 and 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:

This software is provided 'as-is', without any express or implied warranty. In no event will the author be held liable for any damages arising from the use of the software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgement in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source

(continues on next page)

(continued from previous page)

distribution

Mark Adler

madler@alumni.caltech.edu

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 *Connex Micro* libraries, in the *Connex Cert* libraries, and in the Integration Toolkit for AUTOSAR.
- Third-Party Software License:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

12.2 RTI Code Generator (rtiddsgen)

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.
- Version: Release 3.5.2
- Open Source Software License: <https://www.antlr3.org/license.html>

[The BSD License]

Copyright (c) 2010 Terence Parr

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

12.2.2 Apache Commons Lang

- Used in rtiddsgen. 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 as a jar file. The source code is not modified or shipped.
- Source: <https://logging.apache.org/>
- Version: 2.17.1
- Open Source Software License: Apache License Version 2.0 (full text found in the *Appendix*).

12.2.4 Apache Velocity

- This software is included in rtiddsgen. The source code is not modified or shipped.
- Source: <http://velocity.apache.org/>
- Version: 2.3
- Open Source Software License: Apache License Version 2.0 (full text found in the *Appendix*).

12.2.5 Gson

- Portions of rtiddsgen (RTI Code Generator) are built using Gson.
- Version 2.9.1
- Open Source Software License: Apache License Version 2.0 (full text found in the *Appendix*).

12.3 Appendix

12.3.1 Apache License version 2.0, January 2004 (<http://www.apache.org/licenses/>)

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

(continues on next page)

(continued from previous page)

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the

(continues on next page)

(continued from previous page)

Licensors for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained

(continues on next page)

(continued from previous page)

within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the

(continues on next page)

(continued from previous page)

Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Chapter 13

Contact Support

We welcome your input on how to improve *RTI Connect 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.