TWINOAKS COMPUTING INC.
PRACTICAL MIDDLEWARE EXPERTISE

CoreDX DDS

Data Distribution Service

The leading Small Footprint DDS Middleware

Programmer's Guide

Version 3.6

This document describes how to install and use the CoreDX DDS software.

Preface

CoreDX DDS is a small-footprint, high-performance communications middleware compliant with the OMG Data Distribution Service (DDS) standard. CoreDX DDS supports multiple hardware architectures and operating systems, and is intended to facilitate the development of robust, near real-time, highly distributed systems.

This manual describes how to install and use the CoreDX DDS software. It is for developers who want to integrate a high-performance, OMG compliant data distribution middleware service into their application.

How this Guide is Organized

This Programmer's Guide contains a number of Chapters organized in Parts. The first Part provides an overview of the DDS technology and CoreDX DDS. Part 2 provides guidance on installing CoreDX DDS, and walks the reader through creating a simple first CoreDX DDS application.

The next several chapters in Part 3 make up the majority of this document, and go into detail on different aspects of CoreDX DDS features and functionality. This includes: DDS Entities, developing publishing and subscribing applications, Quality of Service (QoS) settings, communication statuses, data instances and samples, data architecture, built-in topics, and the CoreDX DDS transports.

The last few chapters include a discussion of extensions provided by CoreDX DDS such as the logging facility, CoreDX DDS license management, troubleshooting assistance, and finally background about Twin Oaks Computing.

Intended Audience

This document is intended for software developers who are integrating the CoreDX DDS software into their application(s). The guide assumes that the reader is competent in programming languages and software development concepts. CoreDX DDS supports multiple programming languages, and this guide includes examples in C and C++.

Typographic Conventions

Typeface	Meaning	Examples

Courier	Example code	struct StringMsg
		{
		  string msg;
		};

Courier	Example Commands	gunzip –c coredx-2.x.tar.gz

Figure 0-1: Typographic Conventions

Feedback

Twin Oaks Computing welcomes your comments. We are interested in improving our products and we welcome your comments and suggestions. You can provide email feedback about this document to **documents@twinoakscomputing.com**.

Table of Contents

Table of Figures

Table of Tables

Part 1: Introduction

This section provides an introduction of the Data Distribution Service (DDS) and the CoreDX DDS implementation from Twin Oaks Computing, Inc.

Chapter 1 An Introduction to CoreDX DDS

Welcome to CoreDX DDS, a high-performance, small-footprint implementation of the OMG Data Distribution Service (DDS) standard. The CoreDX DDS Data-Centric, Publish-Subscribe messaging infrastructure provides high-throughput, low-latency data communications.

This chapter provides an overview of the Data Distribution Service (DDS), how applications might use DDS to meet their communication requirements, and features of the CoreDX DDS product.

1.1 Why DDS?

Today's enterprise systems, embedded systems, and all systems in between, need flexible, open information systems. Most systems span multiple technologies, hardware platforms, operating systems, and programming languages. In addition, components of these systems have real-time requirements. CoreDX DDS is an open standards-based, communication middleware solution to meet the needs of these real-time distributed systems.

1.2 The case for Middleware

Middleware is a class of software that exists between an application and the Operating System. It provides useful capabilities that are above and beyond those found in standard Operating Systems. In the case of CoreDX DDS, the middleware provides a facility for publish-subscribe communications. Figure 1-1 illustrates where middleware components fit in the application, and how they logically bridge across multiple operating systems and hardware architectures.

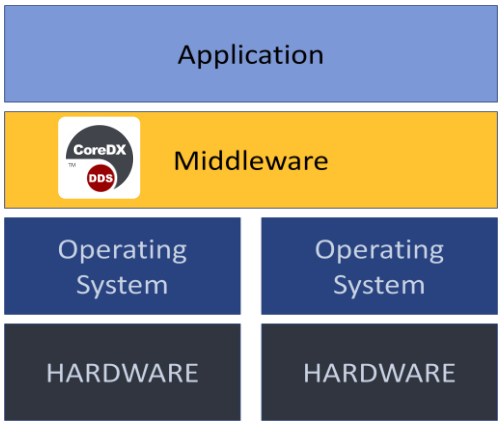

Figure 1-1: Middleware

Applications that employ a communications middleware like CoreDX DDS realize many benefits. The requirements and complexity of data communications in a distributed system are met by the middleware component - leaving developers more time to focus on the important application logic. CoreDX DDS middleware supports many operating systems and hardware architectures - the task of porting complex communications software is already complete.

1.3 The case for Publish Subscribe DDS

Many communication middleware technologies are available. Most are based on a functional model. For example RPC (Remote Procedure Call) and CORBA (Object Request Broker) are two examples of middleware that allow function calls to be distributed across the network between a client and a server. However, these architectures lead to tight coupling between the client and the server; this makes these systems difficult to extend.

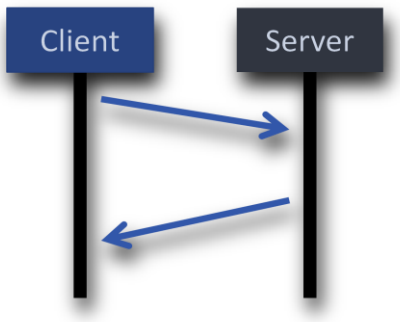

Figure 1-2: Client Server Architecture

The client-server architecture is appropriate for centralized data processing and works well in some systems. However, the drawbacks of this architecture are increased integration costs for new capabilities and potential single point of failure.

An alternative to this approach is the Publish-Subscribe architecture embodied in DDS. This architecture promotes a loose coupling between data producers and data consumers. The architecture is flexible and dynamic; it is easy to adapt and extend systems to changing environments and requirements. Figure 1-3 illustrates the DDS Publish Subscribe architecture where multiple Publishers and Subscribers exchange strongly typed data through a common Topic. The communications are controlled by a Quality of Service model.

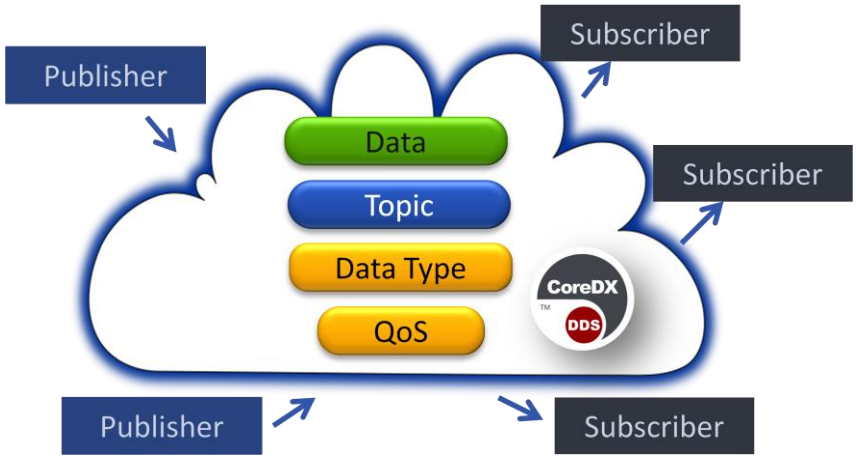

Figure 1-3: Publish Subscribe Architecture

Figure 1-4 is an example of how DDS might be applied in a system. This example has several sources of "raw data", a data processor that performs some processing on the raw data to produce "processed data", several end users working with the processed data, and an administrative user performing analysis, maintenance, or auditing functions.

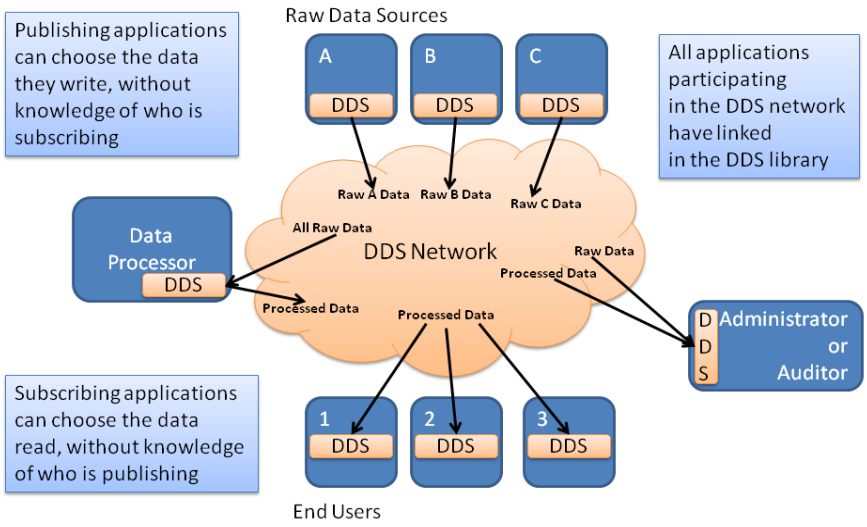

Figure 1-4: Example DDS Usage

In this example, the darker blue boxes represent applications communicating over a DDS network. These applications might be running together on 1 host, or they might be distributed over multiple hosts. A DDS application simply publishes or subscribes to their data, without concern for what, if anything, might be on the other end of its communications. Any of the applications can be dynamically removed (and new applications may be added) without impacting the existing network.

1.3.1 DDS is an Open Standard

DDS is an open specification managed by the Object Management Group (OMG). The OMG is an international, open membership, non-profit organization that develops and manages computer industry specifications. Hundreds of organizations, including software end-users and commercial vendors, make up the OMG. Together they develop and manage many of the standards widely used in the computer industry today. The Data Distribution Service (DDS) is an example of one of the standards managed. Other examples include the Unified Modeling Language (UML), Model Driven Architecture (MDA) and the Common Object Request Broker Architecture (CORBA).

There are several advantages to using a technology that conforms to an open standard, and more advantages if that open standard is managed by an open membership organization like the OMG. First, an open standard promotes interoperability. Anyone, even if they are not connected with the

managing organization, can pick up an Open Standard and write a conforming application. Second, open standards reduce the dependence on a particular vendor. When an open standard product is available from multiple vendors, the consumer can easily change between them. Finally, anyone can join the managing organization and vote on the direction and advancement of the technology. In the case of DDS, this means vendors and users, both public and private, can influence the future of the technology.

1.3.2 DDS is More than a Communications Middleware

The DDS standards specify the mechanism for moving data – a typical communications middleware technology standard. However, DDS is so much more. In addition to communications, DDS provides advanced data management, storage, organization, filtering, and redundancy. With a rich set of features, interoperability across languages, operating systems, hardware platforms, and implementations, DDS provides a robust infrastructure foundation for your small-scale, large-scale, enterprise, embedded, and everything in between software system.

1.3.3 DDS is flexible and scalable

Applications communicating with DDS might be running together on 1 host, or they might be distributed over multiple hosts, each with different architectures and operating systems. Applications using DDS for communications do not need to know the details of where there other applications are residing, or even if they exist.

The *discovery* mechanism built into DDS allows applications to come and go from a DDS network without requiring any changes to the applications or the network. This means a new system can be brought into the network, and start sending or receiving data, without any changes to existing applications.

1.3.4 DDS Features

A DDS application can be a publisher of data, a subscriber of data, or both.

A ***Publisher*** is responsible for data distribution. It may publish data of different data types. The application uses a typed ***DataWriter*** attached to the publisher to communicate the data to be published. Both the Publisher and the DataWriter have a Quality of Service (***QoS***) that affects the behavior of the publication.

A ***Subscriber*** is responsible for receiving published data and making it available to the receiving application. It may receive data of different data types. The application uses a typed ***DataReader*** attached to the subscriber to access the data. Both the Subscriber and DataReader have a QoS that affects the behavior of the subscription. The subscribing application can choose to block waiting for data using ***WaitSets*** or receive data asynchronously, using ***Listeners***.

A Topic fits between publications and subscriptions. Subscriptions must be able to refer to specific publications. A topic fulfills this purpose: it associates a name, a data-type, and a QoS related to the data itself.

When an application wants to publish data of a given type, it must use a Publisher and DataWriter with all the characteristics of the desired publication. When an application wants to subscribe to data of a given type, it must use a Subscriber and DataReader with all the characteristics of the desired subscription.

The following figure depicts the common DDS objects used in exchanging data.

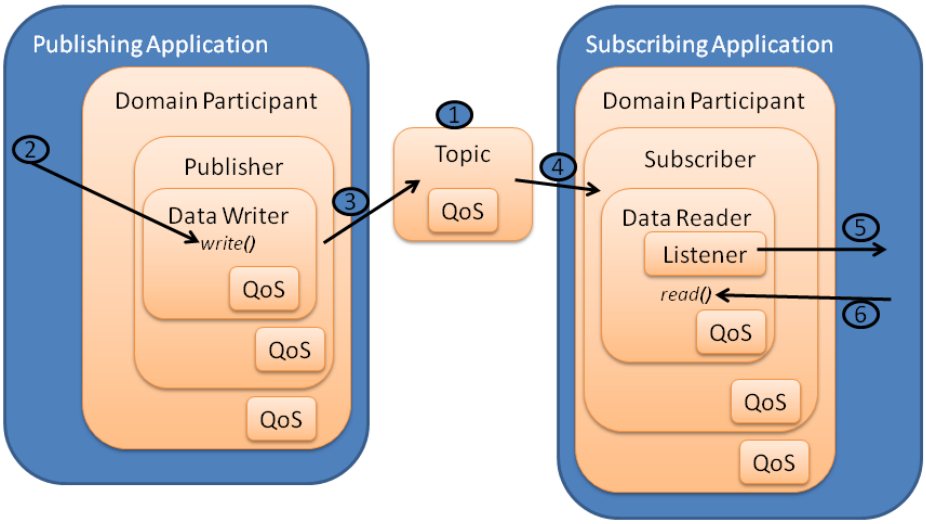

Figure 1-5: DDS Architecture

The following describes the actions depicted in Figure 1-5.

1. *DataReaders* and *DataWriters* are associated with a *Topic*

2. The publishing application calls *DataWriter*::write() to write the data
3. The *Publisher* publishes the data
4. The *Subscriber* receives the data
5. The *Listener* notifies the subscribing application of available data
6. The subscribing application calls *DataReader*::read() to access the data

1.4 The case for CoreDX DDS

The CoreDX DDS provides a quality, high-performance, very small footprint implementation of the DDS standard.

1.4.1 CoreDX DDS is Fast

CoreDX DDS was built from the ground up with performance in mind. The engineering staff at Twin Oaks Computing has a long history of writing and maintaining real-time and near real-time software, and this expertise was used in creating CoreDX DDS. CoreDX DDS is written in 'C' (with additional application language bindings available) for low overhead and memory savings. The CoreDX DDS baseline is tested and enhanced for performance at every step of the development process. The result is a quality DDS implementation with extremely low latency and high throughput capacity.

CoreDX DDS data aggregation, multi-core data pipeline, and low latency event notification provide for throughput in the +900Mbps range and a latencies below 75 usec over a 1Gbps ETHERNET network. But don't take our word for it. The CoreDX DDS release includes source code for example benchmarking applications. Use these examples to compile your own benchmark tests and see how CoreDX DDS performs in your environment, with your data.

1.4.2 CoreDX DDS is Small

The CoreDX DDS product is 100% designed and developed by Twin Oaks Computing to meet the OMG's DDS specification. There is no historical code, no code borrowed from the open source community, no code retrofitted to meet the CoreDX DDS requirements. This allows us to deliver a quality, fully-functional DDS implementation with the smallest footprint. Our entire core library is less than 500 KB, and runs on an Intel Pentium with 640K of memory.

This small library size comes with a proportionally small Line of Code Count, perfect for safety critical applications requiring DO-178B certification.

CoreDX DDS is modular and contains additional run-time memory tuning parameters. Space constrained projects can select components of CoreDX DDS to meet their requirements, and tune those components to reduce unnecessary memory utilization.

1.4.3 CoreDX Uses Multi-Core Technologies

Hardware is moving to multiple core technology. Even embedded processors are shipping with more than one core. This presents a challenge to application developers, because making use of multiple cores requires complex code that is difficult and expensive to develop and maintain. The solution: use a multithreaded communications middleware like CoreDX DDS.

CoreDX DDS was architected from the start to take advantage of multi-core environments. With advanced threading and protections, each CoreDX DDS participant will use a minimum of 3 cores, and typical CoreDX DDS applications will use between 4 and 8 cores. These are single threaded applications, taking advantage of quad-core and higher hardware, just by using CoreDX DDS for data communications.

1.4.4 CoreDX DDS is Self Contained

In order to use CoreDX DDS for communications, the application links in the appropriate CoreDX DDS libraries and that is it. With no daemons and no operating system services that need to be started and maintained, there is no place for data to become "stuck" or for communication states to become corrupted.

1.4.5 CoreDX DDS has Comprehensive Platform Support

With the wide array of language binding, operating system and architecture support, CoreDX DDS runs on a wide variety of platforms, from enterprise servers, to common desktop configurations, to embedded environments and real-time operating systems, to FPGA's and 'bare-metal' configurations. See the *Installation Requirements* section for details on supported platforms.

If you don't find your specific platform listed, just contact us. We offer extensive engineering services, including (often free!) custom ports to new Operating Systems and Architectures, and well as language ports. And with our low line of code count, custom porting is quick and easy.

1.4.6 CoreDX DDS has a great team behind it

A quality DDS implementation is important. But the organization behind the implementation is critical. When you make a commitment to purchase a software product, you are not only obtaining the rights to run the software contained on the installation disk (or downloaded from the web). You are also obtaining support services, training services, and product enhancements for at least the next year.

The staff at Twin Oaks Computing has been developing and supporting large software systems and global software companies for over 50 years. We have worked beside soldiers in Kuwait, sailors onboard aircraft carriers, and other war fighters around the world. We have supported commercial vendors with millions of deployments world-wide. We understand not only the importance of delivering a software product that works, but also the importance of helping companies and their end users make the most of their investment.

We will do the same for you. Give us a call or send us an email. We promise you will receive prompt, friendly, and helpful service.

Part 2: Getting Started

The Getting Started section includes information to establish a development environment and build a simple DDS publisher and subscriber. This provides a quick overview of the development process and the associated tools.

Chapter 2 Installing CoreDX DDS

This chapter explains how to install CoreDX DDS onto your development system.

2.1 Installation Requirements

2.1.1 Supported Architectures and Operating Systems:

Table 2-1: CoreDX DDS Architectures and Operating Systems

Operating System	Architecture	Build Tools
Linux 2.6	x86 (32bit)	gcc-4.3.2 / glibc 2.8
		gcc-3.4.6 / glibc2.3
	x86_64 (64bit)	gcc-4.3.2 / glibc 2.8
		gcc-3.4.6 / glibc2.3
	sun4u	gcc-4.1.2 /glibc 2.8
	ARMv5	gcc-4.1.2
	ARMv7	gcc-4.3.3
	PPC 750, 7400, 440	gcc
	PPC32	gcc
	MIPS 32	gcc-3.4 / uclibc 0.9.29
	Microblaze (soft CPU)	gcc-4.1
Windows XP, Vista, 7, and 8	x86 (32bit)	Visual Studio 2012, 2010, 2008, 2005

Operating System	Architecture	Build Tools
	x86_64 (64bit)	Visual Studio 2012, 2010, 2008
		MinGW / gcc-3.4.5
WinCE, Windows Embedded	X86	Visual Studio 2008
	arm	Visual Studio 2008
Windows 2000	x86 (32bit)	VC6
Solaris 10	i86pc	gcc-3.4.3
		Sun Studio 12
	sun4u	gcc-3.4.3
		Sun Studio 12
QNX 6.4, 6.5	x86 (32bit)	gcc-4.2.4
	ARM v5	gcc
VxWorks 5.5	x86 (32bit)	gcc-4.1.2
	ARM v7	gnu
	PPC405ep	diab
VxWorks 6.6, 6.8, 6.9 (RTP and DKM)	x86 (32bit)	gcc-4.1.2
	PPC32	gcc
NexusWare 12.3	x86	gcc
	PowerPC 440gx	gcc
LynxOS SE 6.0	x86	gcc-4.3

Operating System	Architecture	Build Tools
LynxOS 178	PPC	gcc
INTEGRITY 178B	PPC	gcc
ThreadX	x86	gcc
	PPCe300	gcc
Android	x86	gcc
	ARM v5	gcc
iOS	ARV v7	Apple
DSP/BIOS	TI DSP	TI

2.1.2 Supported Languages and Compilers

Table 2-2: CoreDX DDS Languages and Compilers

Language	Compiler	Compiler Version
C	gcc	multiple
	MS Visual Studio	VS2005, VS2008, VS2010, VS2012, VC6
	MinGW/gcc	3.4.5
	Wind River Diab	
	Sun Studio 12	
C++	g++	3.4.6, 4.3.2
	MS Visual Studio	VS2005, VS2008, VS2010, VS2012, VC6

	Wind River Diab Sun Studio 12	
Java	javac	1.5
C#	Mono (Linux) Visual Studio	2.4 VS2008

2.2 CoreDX DDS Distribution Contents

The CoreDX DDS distribution includes a top-level directory: **coredx-*version***. This top-level directory is referred to throughout this document as **COREDX_TOP*,*** and contains the following files and subdirectories:

COPYRIGHT	:	File(s) describing the Copyright information for this CoreDX DDS baseline
examples	:	Contains example CoreDX DDS applications
host	:	Contains the files required for the HOST (or build) machines for all installed platforms
README	:	File(s) describing the CoreDX DDS version and build
RELEASE_NOTES	:	Information on changes from previous CoreDX DDS versions
scripts	:	Contains helpful, platform independent, scripts
target	:	Contains the files required for the TARGET (or deployment) machines for all installed platforms.

The host and target subdirectories contain the CoreDX DDS libraries and binaries in platform specific directories. This configuration allows you to install CoreDX DDS for multiple platforms in one COREDX_TOP directory. Figure 2-1 shows the directory structure under COREDX_TOP.

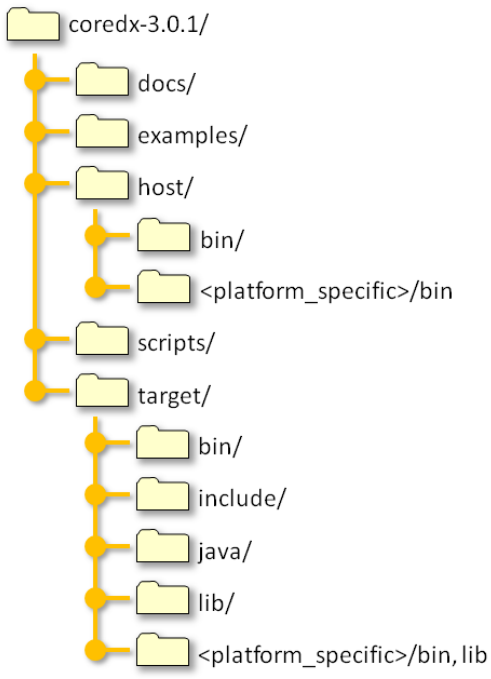

Figure 2-1: CoreDX DDS Directory Structure

The *host* subdirectory contains the HOST tools for CoreDX DDS. The HOST tools include CoreDX DDS DDL compiler and the Twin Oaks Computing license hostid utility for all the installed platforms. The host subdirectory contains two copies of these utilities for each architecture installed. For example, consider the coredx_ddl utility for an x86 Linux system using the gcc4.3 compiler. This utility can be found in two locations:

COREDX_TOP/host/bin/**Linux_2.6_x86_gcc43**_coredx_ddl
COREDX_TOP/host/**Linux_2.6_x86_gcc43**/bin/coredx_ddl

The *target* subdirectory contains the TARGET tools for CoreDX DDS. The TARGET tools include the CoreDX DDS header files, the CoreDX DDS library files, and the Twin Oaks Computing license hostid utility. The CoreDX DDS libraries are located architecture-named subdirectories. For example, consider the DDS library for an x86 Linux system using the gcc4.3 compiler. This library is located:

COREDX_TOP/target/**Linux_2.6_x86_gcc43**/lib/libdds.a

Section 3.3 provides examples for configuring Makefiles to use the CoreDX DDS directory structure.

2.3 CoreDX DDS Installation Procedure

Once you have obtained CoreDX DDS from Twin Oaks Computing (or from the Evaluation CD), unpack the distribution somewhere on your system. For example, on a UNIX system this command will extract the distribution into the current directory:

```
gunzip –c coredx-3.x.x-{platform}.tgz | tar xvf –
```

Or, for Windows:

unzip coredx-3.x.x-{platform}.zip *(It's OK to overwrite files if prompted here.)*

And that's it. There is no additional configuration required.

Customers installing CoreDX DDS for multiple platforms can unpack all the CoreDX DDS releases into the same COREDX_TOP directory. CoreDX DDS uses platform-specific directory names in order to avoid conflicts when working with multiple operating systems and hardware architectures.

2.4 Compiling for a different Target Platform

CoreDX DDS supports cross compiling for a different target platform. For example, if you are developing a VxWorks application, you might be developing (compiling) on a Windows platform. Each platform release of CoreDX DDS contains both the HOST and TARGET tools for one platform, so in the above example, you will require two platform versions of CoreDX DDS, one for the HOST (development) platform, and another for the TARGET (run-time) platform. All platform versions of CoreDX DDS may be installed into the same COREDX_TOP directory.

The DDL compiler (coredx_ddl) is a HOST tool that generates code to be run on the TARGET platform. When the endian of the HOST is different than the endian of the TARGET, it is important to notify the DDL compiler, so that it can generate the correct marshal and un-marshal code for the TARGET platform. This is done using the "-e" option to the DDL compiler. See Chapter 11.10 for additional information on the DDL compiler.

Chapter 3 First CoreDX DDS Application

This chapter describes how to use the CoreDX DDS tools and libraries to integrate basic DDS capabilities into your application. We've provided a sample data type and application that is ported to all CoreDX DDS supported languages and platforms with the distribution (located in COREDX_TOP/examples). You can use these examples or create your own while going though the following steps.

Our example Linux Makefiles were built for `gcc` / `g++`. The examples contain different Makefiles for additional platforms. All example Makefiles use three environment variables: COREDX_TOP, COREDX_HOST, and COREDX_TARGET. The environment script provided with CoreDX DDS releases can help determine the correct settings for these variables (COREDX_TOP/scripts/cdxenv.sh and cdxenv.bat).

3.1 Using a License

Compiling with CoreDX DDS requires a development or evaluation license. Use the environment variable: TWINOAKS_LICENSE_FILE to set the location of your CoreDX DDS license.

For example:

```
linux% export TWINOAKS_LICENSE_FILE=LICENSE_FILE
```

or

```
windows> set TWINOAKS_LICENSE_FILE=LICENSE_FILE
```

3.2 Writing the Application

While CoreDX DDS provides a consistent looking API across all language bindings, there are slight differences in the code generation and compiling instructions.

3.2.1 The 'C' Language Application

Create the Data Definition Language (DDL) file for the data type(s) you will use for communications. The CoreDX DDS DDL syntax is very similar to the OMG IDL syntax for describing data types. Here is the "hello world" example provided with the distribution:

Table 3-1: Sample DDL File

hello.ddl

```
struct StringMsg
{
  string msg;
};
```

Compile the DDL to generate the type specific code using the CoreDX DDS DDL compiler. This requires your TWINOAKS_LICENSE_FILE environment variable be set as described above. Assuming the name of the DDL file is hello.ddl:

```
% COREDX_TOP/host/COREDX_HOST/bin/coredx_ddl -f
hello.ddl
```

The compilation will generate the following files (names are based on the DDL filename):

```
hello.h and .c
helloTypeSupport.h and .c
helloDataReader.h and .c
helloDataWriter.h and .c
```

Create code to publish data of this data type. Our sample Hello World publisher is located in COREDX_TOP/examples/hello_c/hello_pub.c.

Create code to subscribe to data of this data type. Our sample Hello World subscriber is located in COREDX_TOP/examples/hello_c/hello_sub.c.

3.2.2 The 'C++' Language Application

Create the Data Definition Language (DDL) file for the data type(s) you will use for communications. The CoreDX DDS DDL syntax is very similar to the OMG IDL syntax for describing data types. The C++ application uses the same DDL type definition as the C program in the previous section.

Compile the DDL to generate the type specific code using the CoreDX DDS DDL compiler. This requires your TWINOAKS_LICENSE_FILE environment

variable be set as described above. Assuming the name of the DDL file is hello.ddl:

```
% COREDX_TOP/host/COREDX_HOST/bin/coredx_ddl –l cpp –f
hello.ddl
```

The compilation will generate the following files (names are based on the DDL filename):

```
hello.hh and .cc
helloTypeSupport.hh and .cc
helloDataReader.hh and .cc
helloDataWriter.hh and .cc
```

Create code to publish data of this data type. Our sample Hello World publisher is located in COREDX_TOP/examples/hello_cpp/hello_pub.cc.

Create code to subscribe to data of this data type. Our sample Hello World subscriber is located in COREDX_TOP/examples/hello_cpp/hello_sub.cc.

3.2.3 The 'Java' Language Application

Create the Data Definition Language (DDL) file for the data type(s) you will use for communications. The CoreDX DDS DDL syntax is very similar to the OMG IDL syntax for describing data types. The Java application uses the same DDL type definition as the C program in the previous section.

Compile the DDL to generate the type specific code using the CoreDX DDS DDL compiler. This requires your TWINOAKS_LICENSE_FILE environment variable be set as described above. Assuming the name of the DDL file is hello.ddl:

```
% COREDX_TOP/host/COREDX_HOST/bin/coredx_ddl –l java –f
hello.ddl
```

The compilation will generate the following files (names are based on the data type(s) defined in the DDL file):

```
StringMsg.java
StringMsgTypeSupport.java
StringMsgDataReader.java
StringMsgDataWriter.java
```

Create code to publish data of this data type. Our sample Hello World publisher is located in COREDX_TOP/examples/hello_java/HelloPub.java.

Create code to subscribe to data of this data type. Our sample Hello World subscriber is located in COREDX_TOP/examples/hello_java/HelloSub.java.

3.2.4 The 'C#' Language Application

Create the Data Definition Language (DDL) file for the data type(s) you will use for communications. The CoreDX DDS DDL syntax is very similar to the OMG IDL syntax for describing data types. The C# application uses the same DDL type definition as the C program in the previous section.

Compile the DDL to generate the type specific code using the CoreDX DDS DDL compiler. This requires your TWINOAKS_LICENSE_FILE environment variable be set as described above. Assuming the name of the DDL file is hello.ddl:

```
% COREDX_TOP/host/COREDX_HOST/bin/coredx_ddl -l csharp -
f hello.ddl
```

The compilation will generate the following files (names are based on the data type(s) defined in the DDL file):

```
StringMsg.cs
StringMsgTypeSupport.cs
StringMsgDataReader.cs
StringMsgDataWriter.cs
```

Create code to publish data of this data type. Our sample Hello World publisher is located in COREDX_TOP/examples/hello_csharp/hello_pub.cs.

Create code to subscribe to data of this data type. Our sample Hello World subscriber is located in COREDX_TOP/examples/hello_csharp/hello_sub.cs

3.3 Compiling Your Application with CoreDX DDS

Compile your application(s). Our Hello World example creates two applications, one for the publisher and one for the subscriber. This is not necessary, and is completely dependent on your application architecture. Your application will require the objects from the generated type support code above, as well as your publisher and/or subscriber code.

The examples in this section assume a UNIX-based operating system. Compiling for other operating systems may require additional considerations. Please refer to Part 2:Chapter 5*: Advanced Compile Options*

for additional information on compiling CoreDX DDS applications for your specific operating system.

3.3.1 The 'C' Language Application

CoreDX DDS will require the following paths and libraries when compiling your application:

```
Include Path: \
      -I${COREDX_TOP}/target/include
Library Path:  \
      -L${COREDX_TOP}/target/${COREDX_TARGET}/lib
Static Libraries:  -ldds
```

We've provided Makefiles and Visual Studio project files for compiling our examples, and you can use these as a reference for compiling your application. Our Makefiles require three environment variables:

1. COREDX_TOP
2. COREDX_HOST
3. COREDX_TARGET

The COREDX_TOP is a path name to the location of your CoreDX DDS distribution(s). COREDX_HOST and COREDX_TARGET are the platform architectures you are compiling on and compiling for (these can be the same). These values can be set manually, or determined by running the script: COREDX_TOP/scripts/cdxenv.sh or cdxenv.bat.

Our Makefiles will run the DDL compiler to generate the type specific code as well as compile the applications. To compile the Hello World sample application using our Makefile you will need a make program (for example gnu make or Microsoft nmake) and the compiler (for example, gcc or cl.exe) in your path. Then, simply type 'make' (or 'nmake -f NMakefile') in the appropriate directory. This will compile two applications: hello_pub and hello_sub.

3.3.2 The 'C++' Language Application

CoreDX DDS will require the following paths and libraries when compiling your application:

```
Include Path: \
      -I${COREDX_TOP}/target/include
Library Path: \
      -L${COREDX_TOP}/target/${COREDX_TARGET}/lib
```

```
Static Libraries:  -lddscpp -ldds
```

We've provided Makefiles for compiling our examples, and you can use these as a reference for compiling your application. Our Makefile requires three environment variables:

1. COREDX_TOP
2. COREDX_HOST
3. COREDX_TARGET

The COREDX_TOP is a path name to the location of your CoreDX DDS distribution(s). COREDX_HOST and COREDX_TARGET are the platform architectures you are compiling on and compiling for (these can be the same). These values can be set manually, or determined by running the script: COREDX_TOP/scripts/cdxenv.sh or cdxenv.bat.

Our Makefiles will run the DDL compiler to generate the type specific code as well as compile the applications. To compile the Hello World sample application using our Linux Makefile you will need a 'make' program and a C++ compiler in your path. Then, simply type 'make' (or 'nmake -f NMakefile') in the appropriate directory.

This will compile two applications: hello_pub and hello_sub.

3.3.3 The 'Java' Language Application

We've provided scripts for compiling our java examples, and you can use these as a reference for compiling your application. Our scripts require three environment variables:

1. COREDX_TOP
2. COREDX_HOST
3. COREDX_TARGET

The COREDX_TOP is a path name to the location of your CoreDX DDS distribution(s). COREDX_HOST and COREDX_TARGET are the platform architectures you are compiling on and compiling for (these can be the same). These values can be set manually, or determined by running the script: COREDX_TOP/scripts/cdxenv.sh or cdxenv.bat.

Our scripts will run the DDL compiler to generate the type specific code as well as compile the applications. To compile the Hello World sample application using our Linux scripts you will need 'javac' in your path. Then,

simply type 'compile.sh' (or 'compile.bat' for Windows) in the appropriate directory.

This will create a jar file with the two hello applications: HelloPub and HelloSub. We also provide scripts to run these java applications: run_pub.sh and run_sub.sh (.bat scripts are provided for Windows).

3.3.4 The 'C#' Language Application

We've provided Makefiles and Visual Studio project files for compiling our examples, and you can use these as a reference for compiling your application. Our Makefiles require three environment variables:

4. COREDX_TOP
5. COREDX_HOST
6. COREDX_TARGET

The COREDX_TOP is a path name to the location of your CoreDX DDS distribution(s). COREDX_HOST and COREDX_TARGET are the platform architectures you are compiling on and compiling for (these can be the same). These values can be set manually, or determined by running the script: COREDX_TOP/scripts/cdxenv.sh or cdxenv.bat.

Our Makefiles will run the DDL compiler to generate the type specific code as well as compile the applications. To compile the Hello World sample application using our Makefile on Linux you will need the MONO CSharp compiler (gmcs) in your path. Then, simply type 'make' in the appropriate directory. This will compile two applications: hello_pub.exe and hello_sub.exe.

3.4 Running Your Application with CoreDX DDS

You will need at least one environment variable to run your applications with CoreDX DDS:

TWINOAKS_LICENSE_FILE: (refer to *Using a License*, above)

Run your application(s). The sample Hello World has two applications: hello_pub (or hello_pub.exe or HelloPub.class) and hello_sub (or hello_sub.exe or HelloSub.class). The subscriber application (hello_sub) will print out the messages it receives from the publishing application

(hello_pub). The publisher will keep sending out messages until killed. The subscriber will keep listening for messages until killed.

You can run multiple Publishers and multiple Subscribers to immediately see the dynamic nature of the DDS network infrastructure.

Chapter 4 Example CoreDX DDS Applications

CoreDX DDS is bundled with a number of example applications. These applications including working source code and Makefiles to demonstrate how to write, compile, and run CoreDX DDS applications. Most of the example applications are included in the ${COREDX_TOP}/examples directory (exceptions are noted below).

This section includes a description of each of the example programs included in the CoreDX DDS release.

4.1 Environment Setup

All the Makefiles in the CoreDX DDS example application require a few environment variables.

1. COREDX_TOP
2. COREDX_HOST
3. COREDX_TARGET

The COREDX_TOP is a path name to the location of your CoreDX DDS distribution(s). COREDX_HOST and COREDX_TARGET are the platform architectures you are compiling on and compiling for (these can be the same). These values can be set manually, or determined by running the script: COREDX_TOP/scripts/cdxenv.sh or cdxenv.bat.

If you have one CoreDX DDS platform architecture installed, the **cdxenv** script will print out the appropriate commands to set these environment variables. If you have multiple CoreDX DDS platform architectures installed, the **cdxenv** script will list all your platform architectures and prompt you for the correct HOST and PLATFORM architectures before printing the commands to set these environment variables.

Table 4-1: Example - Running cdxenv.sh provides an example of running cdxenv.sh on a Linux machine where multiple CoreDX DDS platform architectures are installed.

Table 4-1: Example - Running cdxenv.sh

cdxenv Example

```
/home/bob/coredx-3.1.0/scripts>  ./cdxenv.sh
   1: Linux_2.6_mips32_gcc34
   2: Linux_2.6_x86_64_gcc43
   3: Linux_2.6_x86_gcc43
   4: NexusWare_ppc440_gcc
   5: SunOS_5.10_sun4u_gcc
   6: VxWorks_6.6_x86_gcc
Please select the HOST platform [2]: 2
   1: Linux_2.6_mips32_gcc34
   2: Linux_2.6_x86_64_gcc43
   3: Linux_2.6_x86_gcc43
   4: NexusWare_ppc440_gcc
   5: NexusWare_x86_gcc
   6: SunOS_5.10_sun4u_gcc
   7: VxWorks_6.6_x86_gcc
Please select the TARGET platform [2]: 7
export COREDX_TOP=/home/bob/coredx_v3.1.0;
export COREDX_HOST=Linux_2.6_x86_64_gcc43;
export COREDX_TARGET=VxWorks_6.6_x86_gcc;
export  LD_LIBRARY_PATH=/home/bob/coredx_v3.1.0/target/VxWor
ks_6.6_x86_gcc/lib
```

4.2 Example 1: The Basic "Hello World" Applications

CoreDX DDS provides three example "Hello World" applications: a 'C' version, a 'C++' version, a 'C#' version, and a Java version. These are simple applications that show the basic usage of the CoreDX DDS entities for sending and receiving data.

Each of these "Hello World" examples contains two applications: one that will publish a "Hello World" message and one that will subscribe to and receive the "Hello World" message.

The hello world examples are located in the examples directory:

```
hello_c/
hello_cpp/
hello_csharp
hello_java/
```

Makefiles are provided for all platform architectures supported by CoreDX DDS, so these applications can be run on a variety of operating systems and languages.

4.3 Example 2: Performance Tests

CoreDX DDS also provides sample performance benchmarking source code for latency and bandwidth benchmarking. These applications provide an example of a more sophisticated CoreDX DDS application. In addition, they allow you to determine the performance of CoreDX DDS with your computers and networking hardware.

The performance examples are located in the examples directory:

```
latency_test\
bwtest\
```

Makefiles are provided for a variety of platform architectures.

4.4 Example 3: Filtering

CoreDX DDS provides an example of using Content Filtered Toipcs in the "dds_filter" example application. Makefiles are provided for a variety of platform architectures.

4.5 Example 4: Dynamic Types

CoreDX DDS provides an example of using Dynamic Types in the "dyntype" example application. Makefiles are provided for a variety of platform architectures.

4.6 Example 5: No Threads

CoreDX DDS provides an example of using CoreDX DDS in a single-threaded mode: the "hello_nothr" example application. This application demonstrates the API for using CoreDX DDS without additional threads.

4.7 Example 5: Shapes Demonstration

The java source code for the CoreDX DDS Shapes demonstration is freely available from the Twin Oaks Computing website (it is not packaged as part of the CoreDX DDS release). The Shapes demonstration provides examples of medium complexity C and Java applications using CoreDX DDS for communications. A specialized version of the CoreDX DDS Shapes demonstration is available for Android platforms.

For additional information and instructions for downloading and using, visit the Twin Oaks Computing website:

http://www.twinoakscomputing.com/coredx/shapes_demo

Chapter 5 Advanced Compile Options

CoreDX DDS includes several libraries that can be used to develop CoreDX DDS applications. Some libraries are required for any CoreDX DDS application. Some libraries include advanced DDS features that should only be used if necessary. Some libraries provide additional debugging information. For all of these libraries, we include a static and dynamic library option (for operating systems that support this distinction).

This section describes the different options available to compile a CoreDX DDS application for different operating system environments.

5.1 Linux and other UNIX-variant Operating Systems

Table 5-1 lists all the libraries provided with the Linux platform release of CoreDX DDS. Other UNIX-variant operating systems may include a sub-set of these libraries (depending on the CoreDX DDS features supported for each particular operating system).

Table 5-1: CoreDX DDS Libraries (UNIX Operating Systems)

Language	library file name	Description
C libraries		
	libdds.a	core library (static library)
	libdds_cf.a	content filter library (static library)
	libdds_cf_log.a	content filter library with logging (static library)
	libdds_cf_log.so	content filter library with logging (dynamic library)
	libdds_cf.so	content filter library (dynamic library)
	libdds_log.a	core library with debug enabled (static library)
	libdds_log.so	core library with debug enabled (dynamic library)

Language	library file name	Description
	libdds.so	core library (dynamic library)
	libdds_dyntype.a	Dynamic Types extension (static library)
	libdds_dyntype_log.a	Dynamic Types extension with logging (static library)
	libdds_dyntype_log.so	Dynamic Types extension with logging (dynamic library)
	libdds_dyntype.so	Dynamic Types extension (dynamic library)
	libcdx_serial_d.a	Serial transport (static library)
	libcdx_tport_lmt.a	Local Machine Transport library
	libcdx_tport_lmt_log.a	Local Machine Transport library with logging
	libcdx_tport_tcp.a	TCP transport library
	libcdx_tport_tcp_log.a	TCP transport library with logging
C++ libraries		
	libdds_cpp.a	C++ language binding (static library)
	libdds_cpp_cf.a	C++ language binding for content filters (static library)
	libdds_cpp_cf_log.a	C++ language binding for content filters with logging (static library)
	libdds_cpp_cf_log.so	C++ language binding for content filters with logging (dynamic library)
	libdds_cpp_cf.so	C++ language binding for content filters (dynamic library)
	libdds_cpp_dyntype.a	C++ Dynamic Types extension (static library)

Language	library file name	Description
	libdds_cpp_dyntype_log.a	C++ Dynamic Types extension with logging (static library)
	libdds_cpp_dyntype_log.so	C++ Dynamic Types extension with logging (dynamic library)
	libdds_cpp_dyntype.so	C++ Dynamic Types extension (dynamic library)
	libdds_cpp_log.a	C++ language binding with logging (static library)
	libdds_cpp_log.so	C++ language binding with logging (dynamic library)
	libdds_cpp.so	C++ language binding (dynamic library)
C# libraries		
	coredx_csharp.dll	C# language binding
	libdds_csharp_log.so	C# language binding – native library with logging
	libdds_csharp.so	C# language binding – native library
Java libraries		
	libdds_java.so	Java language binding – native library
	libdds_java_log.so	Java language binding – native library with logging

CoreDX DDS provides both static (".a") and dynamic (".so") libraries, so the application developer can choose the appropriate type of library to use. All our example applications contain Makefiles that illustrate the use of the different libraries and provide a good reference for CoreDX DDS library usage.

5.1.1 Linking with Static Libraries

A simple CoreDX DDS application written in 'C' requires the core library: libdds.a. The Makefile in the "hello_c" sample application provides an example of using the core library.

A simple CoreDX DDS application written in 'C++' requires the C++ language binding library: libdds_cpp.a ***in addition to*** the core library. The Makefile in the "hello_cpp" sample application provides an example of using the C++ library.

A CoreDX DDS application using content filters must use the content filter library ***in place of*** the core library. A 'C' application will use libdds_cf.a. A 'C++' application will use libdds_cf.a ***and*** libdds_cpp_cf.a. The Makefile in the "dds_filter" sample application provides an example of using the content filter library.

A CoreDX DDS application using dynamic types must use the dynamic type library ***in addition to*** the core library. A 'C' application will use libdds.a ***and*** libdds_dyntype.a. The Makefile in the "dyntype" sample application provides an example of using the dynamic type library. Dynamic types for C++ are not yet supported.

To enable CoreDX DDS logging (refer to *Part 4*:Chapter 14 *CoreDX DDS Logging* for information on CoreDX DDS logging), a CoreDX DDS application must use the logging library ***in place of*** the core library. A 'C' application will use libdds_log.a. A 'C++ application will use libdds_log.a ***and*** libdds_cpp_log.a. There are also content filter and dynamic type libraries with logging enabled.

There is no special configuration required to run a CoreDX DDS application linked with static libraries. Simply set the TWINOAKS_LICENSE_FILE environment variable appropriately as for any CoreDX DDS application.

5.1.2 Linking with Dynamic Libraries

In addition to linking in the correct CoreDX DDS libraries, the LD_LIBRARY_PATH environment variable must be set in order to run dynamically linked CoreDX DDS applications.

To link with dynamic libraries, refer to section 5.1.1*: Linking with Static Libraries* above and replace the ".a" libraries with the ".so" version of the libraries.

To run a CoreDX DDS application compiled with dynamic libraries, the LD_LIBRARY_PATH environment variable must be set, in addition to the TWIN OAKS_LICENSE_FILE environment variable.

5.2 Windows Operating System

Table 5-2 lists all the libraries provided with the Windows platform release of CoreDX DDS.

Table 5-2: CoreDX DDS Libraries (Windows Operating System)

Language	library file name	Description
C libraries		
	dds_cf.dll, .lib dds_cf_d.dll, .lib	content filter library (dynamic library) (debug version)
	dds_cf_log.dll, .lib dds_cf_log_d.dll, .lib	content filter library with logging (dynamic library) (debug version)
	dds_cf_log_static.lib dds_cf_log_d_static.lib	content filter library with logging (static library) (debug version)
	dds_cf_static.lib dds_cf_d_static.lib	content filter library (static library) (debug version)
	dds.dll, lib dds_d.dll, .lib	core library (dynamic library) (debug version)
	dds_dyntype.dll, lib dds_dyntype_d.dll, .lib	Dynamic Types extension (dynamic library) (debug version)
	dds_dyntype_log.dll, lib dds_dyntype_log_d.dll, .lib	Dynamic Types extension with logging (dynamic library) (debug_version)
	dds_dyntype_log_static.lib dds_dyntype_log_d_static.lib	Dynamic Types extension with logging (static library) (with debug)
	dds_dyntype_static.lib dds_dyntype_d_static.lib	Dynamic Types extension (static library) (debug version)

Language	library file name	Description
	dds_log.dll, lib dds_log_d.dll, lib	core library with logging enabled (dynamic library) (with debug)
	dds_log_static.lib dds_log_d_static.lib	core library with logging enabled (static library) (debug version)
	dds_static.lib dds_d_static.lib	core library (static library) (debug version)
C++ libraries		
	dds_cpp_cf.dll, lib dds_cpp_cf_d.dll, .lib	C++ language binding for content filters (dynamic library) (debug version)
	dds_cpp_cf_log.dll, .lib dds_cpp_cf_log_d.dll, .lib	C++ language binding for content filters with logging (dynamic library) (debug version)
	dds_cpp_cf_log_static.lib dds_cpp_cf_log_d_static.lib	C++ language binding for content filters with logging (static library) (debug version)
	dds_cpp_cf_static.lib dds_cpp_cf_d_static.lib	C++ language binding for content filters (static library) (debug version)
	dds_cpp.dll, .lib dds_cpp_d.dll, .lib	C++ language binding (dynamic library) (debug version)
	dds_cpp_log.dll, lib dds_cpp_log_d.dll, .lib	C++ language binding with logging (dynamic library) (debug version)
	dds_cpp_log_static.lib dds_cpp_log_d_static.lib	C++ language binding with logging (static library) (debug version)
	dds_cpp_static.lib dds_cpp_d_static.lib	C++ language binding (static library) (debug version)
	dds_cpp_dyntype.dll, .lib dds_cpp_dyntype_d.dll, .lib	C++ Dynamic Type extension (dynamic library) (debug version)
	dds_cpp_dyntype_static.lib dds_cpp_dyntype_d_static.lib	C++ Dynamic Types extension (static library) (debug version)

Language	library file name	Description
	dds_cpp_dyntype_log.dll, .lib dds_cpp_dyntype_log_d.dll, .lib	C++ Dynamic Types extension with logging (dynamic library) (debug version)
	dds_cpp_dyntype_log_static.lib dds_cpp_dyntype_log_d_static.lib	C++ Dynamic Types extension with logging (static library) (debug version)
C# libraries		
	coredx_csharp.dll	C# language binding
	dds_csharp.dll, .lib	C# language binding – native library
	dds_csharp_log.dll, .lib	C# language binding – native library with logging
Java libraries		
	dds_java.dll, lib	Java language binding – native library (dynamic library)
	dds_java_log.dll, .lib	Java language binding – native library with logging

Both static ("_static.lib") and dynamic (".dll and .lib") libraries are provided, so the application developer can choose the appropriate type of library to use. All our example applications contain Makefiles that illustrate the use of the different libraries and provide a good reference for CoreDX DDS library usage.

5.2.1 Linking with Static Libraries

A simple CoreDX DDS application written in 'C' requires the core library: dds_static.lib. The NMakefile in the "hello_c" sample application provides an example of using the core library.

A simple CoreDX DDS application written in 'C++' requires the C++ language binding library: dds_cpp_static.lib ***in addition to*** the core library. The NMakefile in the "hello_cpp" sample application provides an example of using the C++ library.

A CoreDX DDS application using content filters must use the special content filter library ***in place of*** the core library. A 'C' application will use

dds_cf_static.lib. A 'C++' application will use dds_cf_static.lib ***and*** dds_cpp_cf_static.lib. The NMakefile in the "dds_filter" sample application provides an example of using the content filter library.

A CoreDX DDS application using dynamic types must use the special dynamic type library ***in addition to*** the core library. A 'C' application will use dds_static.lib ***and*** dds_cf_static.lib. Dynamic types for C++ are not yet supported. The NMakefile in the "dyntype" sample application provides an example of using the dynamic type library.

To enable CoreDX DDS logging (refer to Part 4:Chapter 14 *CoreDX DDS Logging* for information on CoreDX DDS logging), a CoreDX DDS application must use the logging library ***in place of*** the core library. A 'C' application will use dds_log_static.lib. A 'C++ application will use dds_log_static.lib ***and*** dds_cpp_log_static.lib. There are also content filter and dynamic type libraries with logging enabled.

There is no special configuration required to run a CoreDX DDS application linked with static libraries. Simply set the TWINOAKS_LICENSE_FILE environment variable appropriately as for any CoreDX DDS application.

NOTE: To build an application linked to static libraries, ***do not include*** /DCOREDX_DLL in the compile flags.

5.2.2 Linking with Dynamic Libraries

Additional compiler flags are required to build an application linked with dynamic libraries. Include the following in the compile flags:

```
/MD
/DCOREDX_DLL
```

Note: the "/MD" specifies dynamically linked applications, while "/MT" is for statically linked applications. To link a dynamic application with debug support, replace "/MD" with "/MDd" and link to the libraries with "_d" in their name.

To link with dynamic libraries, refer to section 5.2.1*: Linking with Static Libraries* above but replace the "_static.lib" libraries with the ".lib" version of the libraries.

There are a few additional rules for linking the correct CoreDX DDS Windows dynamic libraries, due to the nature of Windows dynamic libraries. These rules are described below.

Table 5-3: Windows Dynamic Library Dependencies

If your application is using:	It must also link in the following:
dds_cf.dll, .lib	N/A (standalone library)
dds_cf_log.dll, .lib	N/A (standalone library)
dds_cpp_cf.dll, lib	dds_cf.lib
dds_cpp_cf_log.dll, .lib	dds_cf_log.lib
dds_cpp.dll, lib	dds.lib
dds_cpp_log.dll, lib	dds_log.lib
dds.dll, lib	N/A (standalone library)
dds_dyntype.dll, lib	dds_cf.lib
dds_dyntype._log.dll, lib	dds_cf_log.lib
dds_java.dll, lib	N/A (applications do not link this library)
dds_log.dll, lib	N/A (standalone library)

To run an application linked to CoreDX DDS dynamic libraries, the CoreDX DDS libraries must be in your PATH environment variable. For example:

```
set
PATH=%PATH%;%COREDX_TOP%\target\%COREDX_TARGET%\lib
```

In addition, the TWINOAKS_LICENSE_FILE environment variable must be set correctly.

5.2.3 Dynamic Type Support Library

It may be desirable to create a dynamic library containing the CoreDX DDS generated type support code. This requires some special configuration for Windows.

First, the ***generated type support code*** must be compiled with these additional flags:

```
/MD   (for dynamic applications.  Replace with "/MDd" for debug)
/DCOREDX_DLL
/DCOREDX_DLL_TS
/DCOREDX_DLL_TS_EXPORTS
/LD
```

Note the "/MD" is for dynamically linked applications, while "/MT" is for statically linked applications. To link a dynamic application with debug, replace "/MD" with "/MDd".

The ***application*** linking in the dynamic type support library must be compiled with these additional flags:

```
/MD
/DCOREDX_DLL
/DCOREDX_DLL_TS
```

Then include the dynamic type support library on the link line with the appropriate CoreDX DDS libraries.

To run this resulting application, ensure the generated dynamic type support library is available in your path (in addition to the CoreDX DDS libraries). In addition, the TWINOAKS_LICENSE_FILE must be set appropriately.

Part 3: CoreDX DDS Programming Concepts

This section provides a more detailed examination of the considerations for designing and developing applications that make effective use of the CoreDX DDS middleware. This includes data architecture, Quality of Service configuration, data access patterns, and status event handling.

Chapter 6 DDS Entities

The DDS Standard defines an architecture that represents an object-oriented model of entities that compose the DDS API. These entities serve as the interface between the middleware and the application software. In order to develop a DDS enabled application, a developer must create, interact with, and destroy these DDS entities.

This chapter provides an overview of the DDS Entities and related concepts. Subsequent chapters will provide more details and examples to fully illustrate the API.

6.1 DDS Entity Hierarchy

The primary entities that make up the DDS API are structured in a hierarchy. Each entity in the hierarchy exposes a related set of operations from the API. The programmer interacts with the CoreDX DDS middleware through these DDS entities. For example, the common operations on these entities include creation, destruction, get and set QoS, get and set listeners, get status.

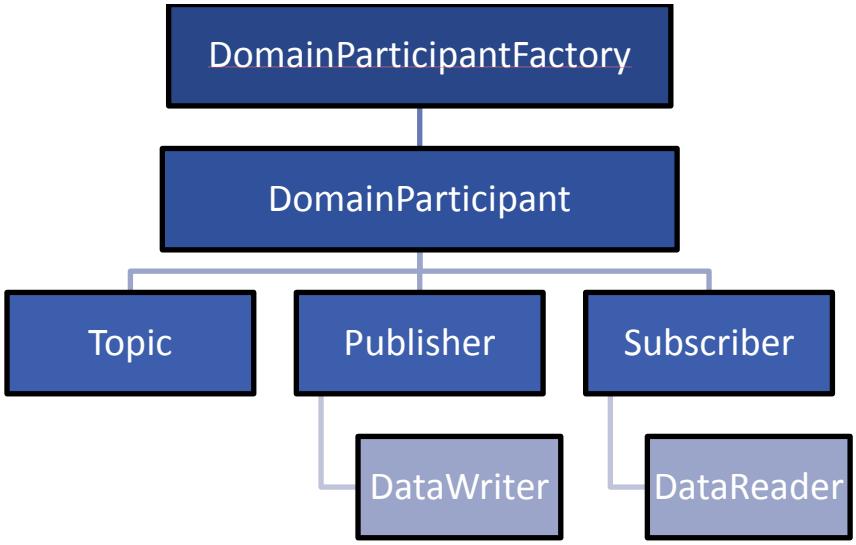

Figure 6-1: DDS Entity Hierarchy

The DomainParticipantFactory is the initial object available to applications. It is a singleton, meaning that only one instance of this object exists within

an application. It is used as a factory to create and delete DomainParticipants.

The DomainParticipant object is the factory for Publishers, Subscribers, and Topics. The DomainParticipant is a container for all of the entities that it creates. The DomainParticipant exists within a 'DOMAIN'. All entities created from a DomainParticipant belong to the same DOMAIN as their parent participant. Entities within a DOMAIN may communicate. Entities in different DOMAINS will not communicate. The DOMAIN specifies a fundamental separation or scope of data.

The Topic entity provides a logical set of homogenous data. The Topic exists within a domain, and has a name and a data type. DataReaders and DataWriters are logically connected by a common topic.

A Publisher entity is a factory for DataWriters. The Publisher is a container for all the DataWriters that it creates. A DataWriter exists within a single Publisher, and is associated with a single Topic. A DataWriter is capable of publishing a single data type that matches its Topic.

A Subscriber entity is a factory for DataReaders. The Subscriber is a container for all the DataReaders that it creates. A DataReader exists within a single Subscriber, and is associated with a single Topic. A DataReader is capable of reading a single data type that matches its Topic.

6.2 DDS Entity Common Operations

Each Entity shares a set of common operations. These operations allow the application to control basic aspects of the entity. For example, an Entity must be enabled before it will participate in communications. If the entity is not enabled automatically by its parent factory, then the enable() method must be called manually.

```
enable()
get_qos()
set_qos()
get_listener()
set_listener()
get_statuscondition()
get_status_changes()
```

6.3 DDS Entity Quality of Service

The behavior of DDS communication is highly configurable. This configuration is performed using Quality of Service policies. Each of these primary DDS Entities accepts Quality of Service parameters to control their behavior. The parent factory maintains a default configuration of QoS for its children. If no QoS is provided a creation time, then these defaults are used. An entity's QoS can be accessed by calling get_qos() on the entity. After an entity is created, its QoS can be changed by directly calling set_qos() on the entity. If the entity is not yet enabled, then any QoS policy can be changed. However, after the entity is enabled, only a subset of QoS policies can be changed on the fly. See the Chapter on QoS Policies for details.

6.4 DDS Status, Listeners, Conditions and WaitSets

Each entity maintains a set of status information. This information represents the occurrence of significant events within the middleware. For example, the "Data Available" or "Publication Matched" statuses are available. The CoreDX DDS middleware supports multiple notification methods to communicate status information to the application. Listener callbacks can be installed, supporting asynchronous notification. Alternatively, the application can initialize a WaitSet and block waiting for a specific set of status conditions. This represents synchronous notification. Finally, the application can choose to poll the middleware using the get_status_changes() method.

Chapter 7 Developing a Publishing Application

This chapter describes the development process for DDS publishing applications.

7.1 Summary of Developing a Publishing Application

The steps for creating a publishing application are as follows:

1. Create or obtain the DDL file for the application data
2. Use the DDL compiler to compile the DDL file. The type-specific support and DataWriter are created as a result of compiling the DDL.
3. Write the publishing application
4. Compile the publishing application

7.2 The DDL File

DDS enabled applications are inherently data-centric. In order for these data-centric applications to perform efficiently, it is necessary to have a well-considered data model, which is implemented in the Data Definition Language (DDL).

For more information on the data types supported by the CoreDX DDS DDL compiler, see *Application Data Types*.

7.3 The Publishing Application

Note: DDS names may be different between the different language bindings. . Some languages use a DDS namespace, while the C language binding augments names to include the namespace as a prefix. For example, in C++ we might reference DDS::DomainParticipantFactory::create_participant(), whereas in C, this would look like DDS_DomainParticipantFactory_create_participant(). For purposes of illustration, the examples in this section are written in C++, and assume that the DDS namespace is in use.

A publishing application must include the generated Type, TypeSupport, and DataWriter header files.

7.3.1 Initialize the DomainParticipantFactory

This is the first step for any application communicating using DDS, and initializes the CoreDX DDS middleware for use.

```
DomainParticipantFactory * dpf =
DomainParticipantFactory::get_instance();
```

7.3.2 Create a DomainParticipant

The DomainParticipant represents the participation of an application in a virtual network, linking all applications that share the same domain ID. Several independent DDS "networks" can coexist in the same physical network without interfering (or even being aware) of each other.

In addition, the DomainParticipant acts as a factory for creating Topics, Publishers (and Subscribers).

When creating a DomainParticipant, you will be able to specify:

Domain ID	The unique identifier for the domain this application will be publishing in
QoS for the DomainParticipant	Describes the QoS for the Domain Participant
Listener	Allows the application to attach listeners to the domain participant.
Listener Status Mask	Sets which listeners are active

To create a DomainParticipant in the 123 domain, using the default DomainParticipant QoS, and no listeners use the following:

```
DomainParticipant * dp = dpf->create_participant( 123,
      PARTICIPANT_QOS_DEFAULT, NULL, 0);
```

7.3.3 Create a Publisher

The publisher is responsible for disseminating (publishing) data. It also acts as a factory for creating DataWriters.

When creating a Publisher, you will be able to specify:

QoS for the Publisher	Describes the QoS for the Publisher
Listeners	Allows the application to attach listeners to the publisher
Listener Status Mask	Sets which listeners are active

To create a Publisher using the default Publisher QoS and no listeners, use the following:

```
Publisher * pub = dp->create_publisher( PUBLISHER_QOS_DEFAULT,
      NULL, 0 );
```

7.3.4 Register a Data Type

In order to publish (or subscribe to) data, the data *type* must be registered in the CoreDX DDS middleware.

To register a data type, use the TypeSupport generated from the DDL. The example code uses a type name: StringMsg. To register this type using the default type name (StringMsg) use the following:

```
StringMsgTypeSupport ts;
ts.register_type( dp, NULL );
```

You can supply an alternate type name by replacing the 2nd argument with a string name.

7.3.5 Create a Topic

The Topic essentially links the publishers of data with the subscribers of data. A Topic is identified by a unique topic name, and a type.

When creating a Topic, you will be able to specify:

Topic Name	Must be unique in the Domain
Type Name	Must already be registered in the Domain
QoS for the Topic	Describes the QoS for the Topic

Listeners	Allows the application to attach listeners to the Topic
Listener Status Mask	Sets which listeners are active

To create a Topic named "HelloTopic" with the "StringMsg" type, default QoS, and no listeners use the following:

```
Topic topic = dp->create_topic( "HelloTopic", "StringMsg",
      TOPIC_QOS_DEFAULT, NULL, 0 );
```

7.3.6 Create a DataWriter

The DataWriter can write data of a specific type. The application uses a type-specific DataWriter to publish data.

When creating a DataWriter, you will be able to specify:

Topic	The Topic to write "on"
QoS for the DataWriter	Describes the QoS for the DataWriter
Listeners	Allows the application to attach listeners to the DataWriter
Listener Status Mask	Sets which listeners are active

To create a DataWriter with the Topic created above, default QoS, and no listeners, use the following:

```
DataWriter * dw = pub->create_datawriter( topic,
      DATAWRITER_QOS_DEFAULT, NULL, 0 );
```

For C++ only: This command creates a "generic" DataWriter (the publisher does not know what type of data will be written). This generic DataWriter will work, however, there is no type checking on the data passed to this generic DataWriter on a write(). In order to have that type checking, use the narrow() method to obtain a type-specific DataWriter:

```
StringMsgDataWriter * sdw = StringMsgDataWriter::narrow(
dw );
```

7.3.7 Write Data

All the necessary pieces are now in place to start publishing (writing) data. There are two methods that can be used for writing:

```
ReturnCode_t retval = sdw->write( data, HANDLE_NIL );
```

And:

```
ReturnCode_t retval = sdw->write_w_timestamp( data,
      HANDLE_NIL, time );
```

The write() method writes the data and uses the current time as the source time stamp. The write_w_timestamp() method allows the application to specify the source time stamp. The source time stamp is sent along with the data, and is located in the SampleInfo on the subscribing end.

Both the write() and write_w_timestamp() methods take an ***instance handle*** argument (the examples above used the empty handle: HANDLE_NIL) and return an instance handle. Each piece of data written is associated in the CoreDX DDS middleware by an instance handle. For more information on instance handles, see the Instances and Sampleschapter. When the data contains a key, calling write() with HANDLE_NIL results in the infrastructure looking up the key data to find the associated instance handle, which takes time. If you are doing multiple writes with data contain the same key (or set of keys), you can optimize the code by calling register() before write(). For example:

```
InstanceHandle_t handle = sdw->register_instance(data);
ReturnCode_t     retval = sdw->write( data, handle );
ReturnCode_t     retval = sdw->write( data, handle );
```

7.4 Available QoS Settings

Every created DDS entity has an associated ***Quality of Service*** (QoS) that can be specified when creating the entity, or later by calling the set_qos() method on that entity. The QoS for each entity is a comprehensive set of configuration policies that affect the behavior of that entity. The middleware defines a default QoS for each entity, which was used in the examples above.

Below is a table listing the available QoS for the publishing entities (DomainParticipant, Publisher, Topic, and DataWriter). This table lists only brief descriptions. A more complete list of QoS Policies and their descriptions can be found in the *Quality* of Service Features chapter.

Table 7-1: QoS Policies for Publishing Entities

QoS Policy	Description	Available To
USER_DATA	Data not used by the CoreDX DDS middleware, the application can use this data for its own purposes.	DomainParticipant, DataWriter
TOPIC_DATA	Data not used by the CoreDX DDS middleware, the application can use this data for its own purposes.	Topic
GROUP_DATA	Data not used by the CoreDX DDS middleware, the application can use this data for its own purposes.	Publisher
DURABILITY	Specifies if published data should be saved for later-joining DataReaders to receive.	Topic, DataWriter
DURABILITY_SERVICE	Not Yet Implemented Specifies configuration for the disabilities: TRANSIENT and PERSISTENT.	Topic, DataWriter
PRESENTATION	Affects how data is presented to the subscribing application. For example, published data can be grouped together such that the DataReaders receive all coherent data together.	Publisher
DEADLINE	Establishes an agreement that the DataWriter will update the data at least once every specified time period.	Topic, DataWriter
OWNERSHIP	Specifies if multiple DataWriters are allowed to write (or update) the same	Topic, DataWriter

	instance of the data, and how these modifications should be handled.	
OWNERSHIP_STRENGTH	Specifies the strength used to arbitrate amount multiple DataWriters writing (or updating) the same instance of the data.	DataWriter
LIVELINESS	Indicates a commitment by the DataWriter to signal it's liveliness to DataReaders in the specified interval. This may mean the DataWriter updates it's samples, or simply asserts it is still alive.	Topic, DataWriter
PARTITION	A logical partition among Topics visible to Publishers and Subscribers. A publisher will only communicate with a subscriber if their Partitions match.	Publisher
RELIABILITY	Indicates the level of reliability offered by the DataWriter.	Topic, DataWriter
DESTINATION_ORDER	Specifies the order in which changes to an instance will be published.	Topic, DataWriter
HISTORY	Specifies if the publishing middleware should keep any (or all) updates to an instance on behalf of existing DataReaders.	Topic, DataWriter
RESOURCE_LIMITS	Specifies the resources the middleware can consume in order to meet the requested QoS.	Topic, DataWriter

ENTITY_FACTORY	Specifies if a factory should automatically enable created entities. If the factory does not automatically enable those entities, the application must specifically enable them before they can be used for publishing or writing data.	DomainParticipantFactory, DomainParticipant, Publisher

7.5 Available Listeners

Listeners are one mechanism allowing the application to be made aware of events and changes in the CoreDX DDS middleware communication status. A listener has a number of methods defined, one for each applicable communication status. The application can define one or more listener methods and attach them to an appropriate DDS entity when the entity is created, or later by using the set_listener() method on the entity.

The table below lists the listeners that can be attached to publishing Entities. A more complete list and description of listeners can be found in the *Communication Status* chapter.

Table 7-2: Listeners for Publishing Entities

Listener	Listener Methods	Description
DataWriterListener	on_offered_deadline_missed()	A deadline offered through the DEADLINE QoS setting was missed.
	on_offered_incompatible_qos()	A DataReader was discovered for the same Topic as this DataWriter, but the QoS requested by that DataReader was incompatible with this DataWriter's QoS.

	on_liveliness_lost()	The liveliness specified in the LIVELINESS QoS was not respected, and DataReaders will consider this DataWriter no longer active.
	on_publication_matched()	A DataReader has been found that matches the Topic and Qos of this DataWriter (or a DataReader that was previously matched is no longer matched.
PublisherListener	(none)	(PublisherListener inherits methods from the DataWriterListener)
TopicListener	on_inconsistent_topic()	Another, different, Topic exists with the same name as this Topic.
DomainParticipant Listener	(none)	(DomainParticipantListener inherits methods from the DataWriterListener, PublisherListener, and TopicListener)

Chapter 8 Developing a Subscribing Application

This chapter describes the development process for DDS subscribing applications.

8.1 Summary of Developing a Subscribing Application

The steps for creating a subscribing application are as follows:

1. Create or obtain the DDL file for the DDS interfaces
2. Use the DDL compiler to compile the DDL file. The type-specific support and DataReader are created as a result of compiling the DDL.
3. Write the subscribing application
4. Compile the subscribing application

8.2 The DDL File

For more information on the data types supported by the CoreDX DDS DDL compiler, see *Application Data Types*.

8.3 The Subscribing Application

Note: Names may be different between the different language bindings. This is because some language bindings use a DDS namespace, and the C language binding augments names to include the namespace as a prefix. For example, in C++ we might reference DDS::DomainParticipantFactory::create_participant(), whereas in C, this would look like DDS_DomainParticipantFactory_create_participant(). For purposes of illustration, the examples in this section are written in C++, and assume that the DDS namespace is in use.

A subscribing application must include the generated Type, TypeSupport, and DataReader header files.

8.3.1 Initialize the DomainParticipantFactory

This is the first step for any application communicating using DDS, and initializes the CoreDX DDS middleware for use.

```
DomainParticipantFactory * dpf =
      DomainParticipantFactory::get_instance();
```

8.3.2 Create a DomainParticipant

The DomainParticipant represents the participation of an application in a virtual network, linking all applications that share the same domain ID. Several independent DDS "networks" can coexist in the same physical network without interfering (or even being aware) of each other.

In addition, the DomainParticipant acts as a factory for creating Topics and Subscribers (and Publishers).

When creating a DomainParticipant, you will be able to specify:

Domain ID	The unique identifier for the domain this application will be publishing in
QoS for the DomainParticipant	Describes the QoS for the Domain Participant
Listener	Allows the application to attach listeners to the domain participant.
Listener status mask	Sets which listeners are active

To create a DomainParticipant in the 123 domain, using the default DomainParticipant QoS, and no listeners use the following:

```
DomainParticipant * dp = dpf->create_participant( 123,
      PARTICIPANT_QOS_DEFAULT, NULL, 0);
```

8.3.3 Create a Subscriber

The subscriber is responsible for receiving data. It also acts as a factory for creating DataReaders. When creating a Subscriber, you will be able to specify:

QoS for the Subscriber	Describes the QoS for the Subscriber
Listeners	Allows the application to attach listeners to the subscriber

Listener Status Mask	Sets which listeners are active

To create a Subscriber using the default Publisher QoS and no listeners, use the following:

```
Subscriber * sub = dp->create_subscriber(
      SUBSCRIBER_QOS_DEFAULT, NULL, 0 );
```

8.3.4 Register a Data Type

In order to subscribe to data, the data *type* must be registered in the CoreDX DDS middleware.

To register a data type, use the TypeSupport generated from the DDL. The example code uses a type name: StringMsg. To register this type using the default type name (StringMsg) use the following:

```
StringMsgTypeSupport ts;
ts.register_type( dp, NULL );
```

You can supply an alternate type name by replacing the 2nd argument with a string name.

8.3.5 Create a Topic

The Topic essentially links the publishers of data with the subscribers of data. A Topic is identified by a unique topic name, and a type.

When creating a Topic, you will be able to specify:

Topic Name	Must be unique in the Domain
Type Name	Must already be registered in the Domain
QoS for the Topic	Describes the QoS for the Topic
Listeners	Allows the application to attach listeners to the Topic
Listener Status Mask	Sets which listeners are active

To create a Topic named "HelloTopic" with the "StringMsg" type, default QoS, and no listeners use the following:

```
Topic topic = dp->create_topic( "HelloTopic",
"StringMsg", TOPIC_QOS_DEFAULT, NULL, 0 );
```

8.3.6 Create a DataReader

The DataReader can read data of a specific type. The application uses a type-specific DataReader to read data. When creating a DataReader, you will be able to specify:

Topic	The Topic to read "on"
QoS for the DataReader	Describes the QoS for the DataReader
Listeners	Allows the application to attach listeners to the DataReader
Listener Status Mask	Sets which listeners are active

To create a DataReader with the Topic created above, default QoS, and no listeners, use the following:

```
DataReader * dr = sub->create_datareader( topic,
      DATAREADER_QOS_DEFAULT, NULL, 0 );
```

An application cannot should cast the DataReader to a type specific DataReader to use it in a type-safe manner. The narrow() operation can be used to obtain the type specific DataReader. For example:

```
StringMsgDataReader * sdr =
      StringMsgDataReader::narrow( dr );
```

8.3.7 Read (or Take) Data

Ultimately, the application will call read() or take() on the DataReader to access data. These operations are non-blocking, and deliver the data that is currently available. The read() and take() operations return an ordered collection of data samples, and their associated sample information, that match the QoS policies set on the Subscriber and DataReader and the parameters passed to read() and take().

The read() and take() operations have a similar signature, and a set of variants that provide the application with additional control over the returned data. The basic read() operation provides the application with access to data managed by the DataReader. After the read() operation, the data is still managed by the DataReader, and is available for access by subsequent read() operations. The take() operation, also provides access to data managed by the DataReader. It differs from read() because data samples accessed by the take() operation are removed from the DataReader, and are not available to subsequent read() or take() operations. As an analogy, read() peeks at the data available in the DataReader while take() actually removes the data from the DataReader.

The basic read() and take() operations both allow the application to specify a filter for view, sample, and instance states. This allows the application to request only those samples that have the requested state. For more information on these states see the *Sample Status Information (SampleInfo)* section.

The variations of read() and take() provided by the API are as follows:

method	Behavior
read_w_condition() **take_w_condition()**	Applies a ReadCondition filter to the samples before returning. The ReadCondition can be a QueryCondition (specialization) which includes an SQL like select statement. This provides for complex filtering of data samples based on status and data content.
read_next_sample() **take_next_sample()**	This accesses a single sample in order as dictated by the QoS settings.
read_instance() **take_instance()**	This accesses the data samples of a particular instance specified as an argument.

read_next_instance() **take_next_instance()**	This provides a mechanism to iteratively access the data samples of all instances. By providing a NIL HANDLE to the first invocation, and then providing the instance handle of the returned samples to subsequent invocations, it is possible to iterate through all instances contained in the DataReader.
read_next_instance_w_condition() **take_next_instance_w_condition()**	This combines filtering capabilities with instance iteration.

8.3.8 Notification Options (Determine When Data is Available)

CoreDX DDS provides a number of status and notifications that are available for the application to receive. An example is the notification that data has been received by the DataReader and is available for the application to read (or take). However, these notification options may also be used to notify the application of other events that may happen within the CoreDX DDS middleware.

8.3.8.1 Using Listeners

Listeners provide asynchronous notification when data is available. There are two listener operations that indicate data is available: the on_data_available() method in the DataReaderListener and the on_data_on_readers() method in the SubscriberListener. The subscribing application attaches a listener to the DataReader or Subscriber, and that listener is invoked when data is available.

Additional information on listeners and example listener code can be found in the *Listeners* section of this manual.

8.3.8.2 Using Conditions

Conditions, when combined with WaitSets provide synchronous notification when data is available by allowing the subscribing application to block until

data is available. There are two types of conditions the subscribing application can use to be notified of available data. The first is a StatusCondition. The DataReader and Subscriber both have a StatusCondition. The DataReader's StatusCondition will trigger when the DATA_AVAILABLE_STATUS on the DataReader changes. The Subscriber's StatusCondition will trigger when the subscriber's DATA_ON_READERS_STATUS changes. The second type of condition is a ReadCondition, which is triggered when data is available on the DataReader. The ReadCondition allows the application to specify additional criteria that must be met before the Condition is triggered, including instance state, sample state, and view state.

Additional information on Conditions and WaitSets, along with example code can be found in the *Conditions and WaitSets* section of this manual.

8.3.8.3 Using Polling

The application can choose to poll for data, rather than blocking or using callbacks. When polling for data, the application calls DataReader::read() or take() operation in a loop. If there is data available, these methods return DDS::RETCODE_OK, otherwise they return DDS::RETCODE_NO_DATA.

8.4 Sample Status Information (SampleInfo)

Calls to any of the read() variants described above return one ore more samples and corresponding *SampleInfo* structures. The *SampleInfo* structure contains metadata about the received sample and includes the following information:

8.4.1 sample_state

The **sample state** is the state of the data *sample.* Valid states are: READ and NOT_READ.

The sample state is READ if this DataReader has read this sample previously, otherwise the state is NOT_READ.

8.4.2 view_state

The **view state** indicates this DataReader's view of the data *instance.* Valid states are: NEW and NOT_NEW.

The view state is NEW if this DataReader has never read a sample from this instance, otherwise the state is NOT_NEW. The view state can also be NEW if this is the first sample received since the instance was disposed.

8.4.3 instance_state

The **instance state** is the state of the instance. Valid states are: ALIVE, NOT_ALIVE_DISPOSED, and NOT_ALIVE_NO_WRITERS.

The instance state is ALIVE if there is at least one DataWriter actively writing samples on this instance. The instance state is NOT_ALIVE_DISPOSED if the instance was explicitly disposed by a DataWriter. The instance state is NOT_ALIVE_NO_WRITERS if there are no DataWriters actively writing this instance.

8.4.4 source_timestamp

The **source timestamp** is the timestamp provided by the DataWriter at the time the sample was produced.

8.4.5 instance_handle

The **instance handle** is a unique identifier for this instance.

8.4.6 publication_handle

The **publication handle** is a unique identifier for the DataWriter who wrote this sample.

8.4.7 disposed_generation_count

The **disposed generation count** is a count of the number of times the instance has come alive *after* being disposed. In other words, any time the instance state changes from NOT_ALIVE_DISPOSED to ALIVE.

This count can be used to determine the number of times an instance has been disposed. Initially, it is 0.

8.4.8 no_writers_generation_count

The **no writers generation count** is a count of the number of times a DataWriter has started writing data on the instance *after* being declared NOT_ALIVE_NO_WRITERS. In other words, any time the instance state changes from NOT_ALIVE_NO_WRITERS to ALIVE.

This count can be used to determine the number of times an instance has not been alive due to no active readers. Initially, it is 0.

8.4.9 sample_rank

The **sample rank** is the number of samples in this instance that follow this one in the current read (or take) collection.

The sample rank can be used to determine the 'sample age' of the current sample, relative to the number samples for the instance in the returned sample set.

8.4.10 generation_rank

The **generation rank** is the number of times this instance has transitioned from not-alive to alive in the time between the reception of this sample and the latest sample for this instance in the current read (or take) collection.

The generation rank can be used to determine the 'generation age' of the current sample, relative to the number of samples for this instances in the returned sample set.

8.4.11 valid_data

The **valid data** flag indicates the sample data associated with this SampleInfo is valid. Valid values are zero (FALSE) and non-zero (TRUE).

The valid data flag is set to true when a data sample is received. The valid data flag is set to false when an unregister or dispose command is received.

The read() operation will deliver "access" to the corresponding data. The data is not removed from the middleware, and a subsequent read() or take() operations can provide the same data. The take() operation will deliver the corresponding data to the application. After a take() the data is no longer available and subsequent reads() or takes() will not provide the same data.

The application can use a variety of methods for determining data availability. [These are the same methods for obtaining other DDS statuses]. They include: asynchronous, synchronous, and polling. These three models are described here

8.5 Additional Subscriber / DataReader Features

8.5.1 Filtering Data

There are two basic options for filtering received data.

1. Filter data that is received by the DataReader (filtered data is never available to the application).
2. Filter data as it is read by the application (filtered data is still available in the DataReader for future reads or takes by the application).

A ContentFilteredTopic or the Time Based Filter QoS policy is used to achieve the first option. A DataReader created on a ContentFilteredTopic will not store the filtered data, and so it is never available to the application. Refer to the *Content Filtered Topics* section for additional information on ContentFilteredTopics. Similarly, data filtered by a configured Time Based Filter QoS policy is not added to the DataReader cache, and so it is never available to the application.

The second option may be achieved by using the read_w_condition() (or take_w_condition()) API, or by using a WaitSet with a read condition attached. Both the read_w_condition() / take_w_condition() API and WaitSets allow filtering using *Read Conditions* or *Query Conditions*.

Read Conditions allow the application to filter by the state and view of the data in the DataReader cache. *Read Condition* filters parameters include:

- Sample State: has the data sample been 'read' or not
- View State: is the data sample newly received since the application last accessed the data cache (via a read() or take() operation)
- Instance State: is the instance *alive*, *disposed*, or *unregistered* (see xyz for additional information on Instances).

Query Conditions allow the application to filter on the data contents of each sample. These conditions are provided as an SQL-like query string, and only data that matches the specified query is returned to the application. Refer to the *Content Filtered Topics* section for additional information on the query syntax.

8.5.2 Wait for Historical Data

DataReaders with a Durability QoS policy configured to *Transient Local, Transient, or Persistent* may receive historical data published before this DataReader was enabled. The DataReader provides an API that will block the application until all available historical data has been received:

```
DataReader::wait_for_historical_data( Duration_t
max_wait)
```

When this method is invoked, the application will block until all historical data (all previously published data samples) have been received and are available for the application to read or until max_wait has expired, whichever occurs first.

This method is not applicable when the DataReader's Durability QoS policy is configured to *Volitile*; in this case, wait_for_historical_data() will return immediately.

8.6 QoS Policies

Every created DDS entity has an associated ***Quality of Service*** (QoS) that can be specified when creating the entity, or later by calling the set_qos() method on that entity. The QoS for each entity is a comprehensive set of configuration policies that affect the behavior of that entity. The middleware defines a default QoS for each entity, which was used in the examples above.

Below is a table listing the available QoS for the subscribing entities (DomainParticipant, Subscriber, Topic, and DataReader). This table lists only brief descriptions. A more complete list of QoS Policies and their descriptions can be found in the *Quality* of Service Features chapter.

Table 8-1: QoS Policies for Subscribing Entities

QoS Policy	Description	Available To
USER_DATA	Data not used by the CoreDX DDS middleware, the application can use this data for its own purposes.	DomainParticipant, DataReader
TOPIC_DATA	Data not used by the CoreDX DDS middleware, the application can use this	Topic

	data for its own purposes.	
GROUP_DATA	Data not used by the CoreDX DDS middleware, the application can use this data for its own purposes.	Subscriber
DURABILITY	Specifies if the DataReader would like to receive older data that was published before the DataReader came online (in other words, the history data).	Topic, DataReader
DURABILITY_SERVICE	Not Yet Implemented Specifies configuration for the disabilities: TRANSIENT and PERSISTENT.	Topic, DataReader
PRESENTATION	Affects how data is presented to the subscribing application. For example, published data can be grouped together such that the DataReaders receive all coherent data together.	Subscriber
DEADLINE	Establishes an expectation that the publisher of data will updated the data at least once every specified time period.	Topic, DataReader
OWNERSHIP	Specifies if multiple DataWriters are allowed to write (or update) the same instance of the data, and how these modifications should be handled.	Topic, DataReader
LIVELINESS	Indicates an expectation of the DataReader that the DataWriter will signal it's liveliness in the specified interval.	Topic, DataReader
PARTITION	A logical partition among Topics visible to Publishers and Subscribers. A publisher will only communicate with a subscriber if their Partitions match.	Subscriber

RELIABILITY	Indicates the level of reliability expected of all DataWriters.	Topic, DataReader
DESTINATION_ORDER	Specifies the order in which changes to an instance will be ordered by the Subscriber.	Topic, DataReader
HISTORY	Specifies if the subscribing middleware should keep any (or all) updates to an instance (history).	Topic, DataReader
RESOURCE_LIMITS	Specifies the resources the middleware can consume in order to meet the requested QoS.	Topic, DataReader
ENTITY_FACTORY	Specifies if a factory should automatically enable created entities. If the factory does not automatically enable those entities, the application must specifically enable them before they can be used for receiving data.	DomainParticipantFactory, DomainParticipant, Subscriber

8.7 Available Listeners

Listeners are one mechanism allowing the application to be made aware of events and changes in the CoreDX DDS middleware communication status. We illustrated an example of one kind of listener above, the DataReaderListener, used for receiving data. There are additional listeners available to a subscribing application. The application can define one or more listener methods and attach them to an appropriate DDS entity when the entity is created, or later by using the set_listener() method on the entity.

The table below lists the listeners that can be attached to a subscribing application. A more complete list and description of listeners can be found in the *Communication Status* chapter.

Table 8-2: Listeners for Subscribing Entities

Listener	Listener Methods	Description
DataReader Listener	on_requested_deadline_missed()	The deadline this DataReader was expecting through its QoS DEADLINE was missed.
	on_requested_incompatible_qos()	A DataWriter has been discovered that has a QoS configuration incompatible with this DataReader's QoS
	on_liveliness_changed()	One or more of the DataWriters this DataReader was receiving samples data from has changed liveliness (either became ACTIVE: actively writing samples, or INACTIVE)
	on_subscription_match()	A DataWriter has been discovered that matches the Topic and has a compatible QoS configuration to this DataReader
	on_sample_rejected()	A received sample has been rejected by this DataReader because a RESOURCE_LIMITS QoS setting has been exceeded
	on_data_available()	New information (data sample(s) or sample information) is available
	on_sample_lost()	A sample has been lost (not received by this DataReader)

Listener	Listener Methods	Description
Subscriber Listener	on_data_on_readers()	Data has been received and is available on one or more DataReaders attached to this Subscriber. (SubscriberListener also inherits methods from the DataReaderListener)
Topic Listener	on_inconsistent_topic()	Another, different, Topic exists with the same name as this Topic.
DomainParticipant Listener	(none)	(DomainParticipantListener inherits methods from the DataReaderListener, SubscriberListener and the TopicListener)

Chapter 9 Topics

9.1 Overview

Topics connect publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A Topic is meant to fulfill that purpose: it associates a name, a data-type, and QoS related to the data itself.

Each topic corresponds to one data type. However, several topics may refer to the same data type.

Topics have a ***Quality of Service*** (QoS) that describes the data written to that topic. The topic QoS can be specified when creating the topic, or later by calling the set_qos() operation on the topic. The QoS defined for the topic is not used by CoreDX DDS, but may be used by the application as a hint for the QoS of the corresponding DataReaders and DataWriters. Additional information on QoS policies can be found in the *Quality* of Service Features chapter.

There are several variations of a topic. The *base class* for all topics is a ***TopicDescription***. The TopicDescription contains the topic name and data-type name. There are three (3) variations of a TopicDescription. They are listed in Table 9-1.

Table 9-1: Topic Variants

Topic Variants	Description
Topic	The basic form of a TopicDescription, it contains a description of the data to be published and subscribed to, including QoS and Listeners.
ContentFilteredTopic	This topic allows for content-based subscriptions, that is, a subscription that receives a subset of the data published based on a SQL-like query condition.

MultiTopic	This topic allows for combining and filtering data from several topics. CoreDX DDS currently does not support ***MultiTopics***.

Topics are created using one of the create_topic() operation variations provided from the DomainParticipant.

9.2 Built-In Topics

The CoreDX DDS infrastructure manages a set of *built-in topics*. These topics are created when a DomainParticipant is initialized, and keep track of discovery information about other DDS participants, Topics, DataReaders, and DataWriters. This information is necessary for the DDS discovery to work properly, and may also be useful to applications that want to react to this discovery information.

Table 9-2 lists the built-in topics and their associated data types.

Table 9-2: Built-in Topics

Built-in Topic Name	Data Type Name	Description
DCPSParticipant	DDS::ParticipantBuiltinTopicData	Samples are description of DDS participants that have been discovered by this DomainParticipant
DCPSTopic	DDS:: TopicBuiltinTopicData	Samples are descriptions of Topics discovered by this DomainParticipant
DCPSPublication	DDS:: PublicationBuiltinTopicData	Samples are descriptions of DataWriters discovered by this DomainParticipant
DCPSSubscription	DDS:: SubscriptionBuiltinTopicData	Samples are descriptions of DataReaders discovered by this

DomainParticipant

In general, the built-in data types hold information about the discovered entity's QoS configuration, along with other useful information. For a detailed description of these built-in data types, refer to the dds_builtin.h or dds_builtin.hh header files in the COREDX_TOP/target/COREDX_TARGET/include/dds directory.

Each built-in topic has a type-specific DataReader associated with it (DCPSParticipantDataReader, etc.). The application can use these DataReaders to access the data and statuses from the built-in topics in the same way any user defined DataReader would do this.

To get access to these built-in DataReaders, the application can call

```
DomainParticipant::get_builtin_subscriber()
```

This Subscriber can be used to access specific built-in data readers by calling

```
Subscriber::lookup_datareader(topic_name)
```

and using the appropriate topic name from

Table 9-2.

The built-in topics use data types specified in the DDS standard for communicating discovery data. The following tables illustrate the data type of each of the built-in topics.

Table 9-3: Participant Built-in Data Type

ParticipantBuiltinTopicData

```
struct ParticipantBuiltinTopicData
{
  DDS_KEY   BuiltinTopicKey_t    key;
            UserDataQosPolicy    user_data;
};
```

Table 9-4: Topic Built-in Data Type

TopicBuiltinTopicData

```
struct TopicBuiltinTopicData
{
  DDS_KEY   BuiltinTopicKey_t          key;
            string                     name;
            string                     type_name;
            DurabilityQosPolicy        durability;
            DurabilityServiceQosPolicy durability_service;
            DeadlineQosPolicy          deadline;
            LatencyBudgetQosPolicy     latency_budget;
            LivelinessQosPolicy        liveliness;
            ReliabilityQosPolicy       reliability;
            TransportPriorityQosPolicy transport_priority;
            LifespanQosPolicy          lifespan;
            DestinationOrderQosPolicy  destination_order;
            HistoryQosPolicy           history;
            ResourceLimitsQosPolicy    resource_limits;
            OwnershipQosPolicy         ownership;
            TopicDataQosPolicy         topic_data;
};
```

Table 9-5: Publication Built-in Data Type

PublicationBuiltinTopicData

```
struct PublicationBuiltinTopicData
{
  DDS_KEY   BuiltinTopicKey_t          key;
            BuiltinTopicKey_t          participant_key;
            string                     topic_name;
            string                     type_name;
            DurabilityQosPolicy        durability;
            DurabilityServiceQosPolicy durability_service;
            DeadlineQosPolicy          deadline;
            LatencyBudgetQosPolicy     latency_budget;
            LivelinessQosPolicy        liveliness;
```

```
            ReliabilityQosPolicy        reliability;
            LifespanQosPolicy           lifespan;
            UserDataQosPolicy           user_data;
            OwnershipQosPolicy          ownership;
            OwnershipStrengthQosPolicy  ownership_strength;
            DestinationOrderQosPolicy   destination_order;
            PresentationQosPolicy       presentation;
            PartitionQosPolicy          partition;
            TopicDataQosPolicy          topic_data;
            GroupDataQosPolicy          group_data;
};
```

Table 9-6: Subscription Built-in Data Type

SubscriptionBuiltinTopicData

```
struct SubscriptionBuiltinTopicData
{
  DDS_KEY   BuiltinTopicKey_t           key;
            BuiltinTopicKey_t           participant_key;
            string                      topic_name;
            string                      type_name;
            DurabilityQosPolicy         durability;
            DeadlineQosPolicy           deadline;
            LatencyBudgetQosPolicy      latency_budget;
            LivelinessQosPolicy         liveliness;
            ReliabilityQosPolicy        reliability;
            OwnershipQosPolicy          ownership;
            DestinationOrderQosPolicy   destination_order;
            UserDataQosPolicy           user_data;
            TimeBasedFilterQosPolicy    time_based_filter;
            PresentationQosPolicy       presentation;
            PartitionQosPolicy          partition;
            TopicDataQosPolicy          topic_data;
            GroupDataQosPolicy          group_data;
};
```

9.3 Content Filtered Topics

A ContentFilteredTopic is a specialization of TopicDescription that allows the subscribing application to describe a subscription where it will only see a subset of the data published, based on a defined content filter. The filter is

an SQL like statement. The ContentFilteredTopic is associated with another known Topic and applies a filter to the data available on that related topic.

ContentFilteredTopics are created by a DomainParticipant, just like normal Topics.

```
DomainParticipant::create_contentfilteredtopic()
```

This method call has additional parameters to specify which topic the ContentFilteredTopic is associated with, the SQL query expression, and parameters (if any) for use in evaluating the filter expression.

Table 9-7: create_contentfilteredtopic() parameters

Parameter	Description
Topic	Related topic. The ContentFilteredTopic presents a filtered subset of data available on the related topic.
filter_expression	SQL like condition expression
filter_parameters	String sequence of parameters used in the **filter_expression**.

The **filter_expression** must be a valid SQL WHERE clause (without the **WHERE** keyword). For example "x<=4". The filter expression refers to structure members in the application defined data type associated with the related Topic. For embedded structures, the naming convention uses a dot ('.') to separate field names. For example, "time.sec > 10". Table 9-8 lists all the operators available when constructing the condition.

Table 9-8: Valid Condition Operators for Content Filters

Operator	Description
=	Equals
<>	Not equals
>=	Greater than or equal

Operator	Description
>	Greater than
<=	Less than or equal
<	Less than
NOT, not	Not operator
( )	Parenthesis are used for nesting conditions
AND, and	And operator
OR, or	Or operator
IN, in	In operator for matching a value to something in a list
BETWEEN, between	For future use: the between operator is not yet supported by CoreDX DDS.
LIKE, like	For future use: the like operator is not yet supported by CoreDX DDS.

The filter expression can also contain references to parameters, present in the **filter_parameters** argument. Parameter references take the form "%<number>". The number is an index into the filter_parameters sequence, and starts at zero. For example "%2" refers to the third parameter in the filter_parameters sequence.

Once created ContentFilteredTopics can be used by a DataReader just like a normal Topic. The filter expression is static, and cannot be changed after the ContentFilteredTopic is created; however, filter parameters can be changed on the fly with a call to

```
ContentFilteredTopic::set_expression_parameters()
```

Content Filter expressions can contain references to basic data types. For example data members of type int, short, long, and string are all valid field types for a filter expression; but, sequences, arrays, or unions are not.

9.3.1 Content Filter Example

The full code for a content filter example can be found in the examples directory of the CoreDX DDS release.

Table 9-9: Creating a ContentFilteredTopic

Creating a ContentFilteredTopic (C language)

```
DDS_DomainParticipant    dp;
DDS_Subscriber           sub;
DDS_Topic                top;
DDS_ContentFilteredTopic cftop;
DDS_TopicDescription     td;
DDS_DataReader           dr;
DDS_StringSeq            cf_params;

dp  = DDS_DomainParticipantFactory_create_participant( 1,
          DDS_PARTICIPANT_QOS_DEFAULT, NULL, 0);
sub = DDS_DomainParticipant_create_subscriber(dp,
          DDS_SUBSCRIBER_QOS_DEFAULT, NULL, 0);
MyTypeTypeSupport_register_type( dp, "topic_type" );
top   = DDS_DomainParticipant_create_topic(dp, "topic_name",
          "topic_type", DDS_TOPIC_QOS_DEFAULT, NULL, 0);

/* BUILD A CONTENT_FILTERED TOPIC */
cftop = DDS_DomainParticipant_create_contentfilteredtopic(dp,
          "cf_topic_name", top,   "x<%0", NULL);

/* parameters can be specified/modified after creation */
INIT_SEQ(cf_params);
seq_set_size(&cf_params, 1);
seq_set_length(&cf_params, 1);
cf_params._buffer[0] = "5";
DDS_ContentFilteredTopic_set_expression_parameters(cftop,
          &cf_params);

td  = DDS_Topic_TopicDescription((DDS_Topic)cftop);
dr  = DDS_Subscriber_create_datareader(sub, td, &dr_qos,
          NULL, 0);
if (!dr)
  printf("FAILED to create DR!\n");
```

9.3.2 Configuring Content Filters

When a DataReader uses a content filter to filter the received data, the filter expression is communicated to any matching DataWriter(s). This allows CoreDX DDS to filter data either at the DataWriter or the DataReader.

The DataReader's content filter is always enabled. This ensures the specified data is always filtered, even when the DataReader is matched with a DataWriter that does not support writer-side-filtering (for example, this may happen when interoperating with another DDS implementation).

DataWriter content filtering is configurable by the DataWriter QoS policy: RTPSWriterQosPolicy.apply_filters (true or false value).

DataWriter filtering is enabled by default. This means the DataWriter will write data to a DataReader only when it passes the DataReader's content filter (or if the DataReader does not have a content filter configured). This configuration can reduce the 'work' performed by the DataReader to apply the filter, but also means the DataWriter must unicast all samples that are filtered by at least 1 matched DataReader.

When the DataWriter is configured to NOT apply filters, the DataWriter will always multicast written data samples, allowing the DataReader to apply the filter.

9.3.3 Compiling an application with Content Filters

CoreDX DDS provides a separate library that contains the Content Filter capability. This allows us to keep the basic CoreDX DDS library extremely small. To compile an application that uses content filters, change the library line in your Makefile, replacing libdds with **libdds_cf**. For the C++ language binding, you will also need to replace libdds_cpp with libdds_cpp_cf. CoreDX DDS does not yet support content filters in the Java API.

9.4 Multi Topics

A MultiTopic is a specialization of TopicDescription that allows the application to describe a subscription that combines, filters, and orders data from multiple Topics. This is similar to the **JOIN** statement in SQL and Relational Database Systems.

CoreDX DDS currently does not support multi topics.

Chapter 10 Instances and Samples

Data is the core of any communications middleware, and it is especially important to a data-centric, publish-subscribe middleware like CoreDX DDS. This chapter describes how the CoreDX DDS middleware handles and classifies the data, and how the data is packaged and communicated between the application and the CoreDX DDS middleware.

10.1 Overview

Each Topic is attached to a DDS ***data type***. Only data of that type may be published on the Topic. The DDS data type is always a structure, which can be made up of virtually any user defined type. The following is an example of a DDS data type.

```
struct HelloMessage {
      long              time_sent;
      long              sender_id;
      string            sender_name;
      string            msg;
      sequence<string>  msg_history;
}
```

The type name for the above DDS data type is "HelloMessage".

Additional information on the generating DDS data types can be found in the *Application Data Types* chapter.

The CoreDX DDS middleware classifies data into ***samples*** and ***instances***. A ***sample*** is data of the appropriate DDS data type that has been published to a DDS Topic. The following is a possible ***sample*** of the HelloMessage data type:

```
123456789
42
"Bob The Builder"
"Hello out there!"
<empty sequence>
```

The application developer, when creating a DDS data type, can specify one or more attributes of the DDS data type as a ***key***. The CoreDX DDS middleware uses those key attributes to organize the published data. Going back to the HelloMessage example above, an application developer might specify the "sender_id" field as the key for the HelloMessage data type.

A publishing application might write the following four (4) ***samples***:

```
123456789
42
"Bob The Builder"
"Hello out there!"
<empty sequence>
```

```
223456789
45
"Wendy"
"Busy day today"
<empty sequence>
```

```
323456789
42
"Bob The Builder"
"Can We Build It?"
<empty sequence>
```

```
423456789
12
"Scoop"
"Yes we can!"
<empty sequence>
```

If the Data Type defines the "sender_id" field as the key for the HelloMessage data type, then in the 4 samples published, there are 3 unique keys. The CoreDX DDS middleware classifies all samples with the same key value to be one (1) ***instance***. In this example, 3 instances have been published, with the following keys: 42, 45, and 12. This is depicted in Table 10-1.

Table 10-1: Instance Example

Instance 1, Key=42 2 samples	Instance 2, Key=45 1 sample	Instance3, Key=12 1 sample
123456789 42 "Bob The Builder" "Hello out there!" <empty sequence>	223456789 45 "Wendy" "Busy day today" <empty sequence>	423456789 12 "Scoop" "Yes we can!" <empty sequence>
323456789 42 "Bob The Builder" "Can We Build It?" <empty sequence>		

The CoreDX DDS middleware stores and manages data ***samples*** and ***instances***.

10.2 Publishing Data

On the publishing side of DDS communications, ***samples*** represent data that is sent to DataReaders. Samples are created for every write(), unregister(), and dispose() call made by the application. Each sample written is associated with a particular instance. In general, samples and instances are stored by the DataWriter until they are delivered to all appropriate DataReaders, at which point the samples and instances may be removed. The specific rules for maintaining *samples* in the DataWriter are different from the rules for managing *instances.* For this reason, it is possible for all samples on an instance to be removed from the DataWriter, while the instance remains (with no associated samples). In contrast, it is *not* possible to remove an instance from the DataWriter while any samples associated with it remain.

Data ***instances*** are used to manage several DataWriter QoS policies. ***Instances*** allow the application to set *Deadlines*, keep *History*, manage *Ownership*, and follow *Resource Limits*.. For additional information on these QoS policies, see the *Quality of Service Features* Chapter.

10.3 Subscribing to Data

On the subscribing side of CoreDX DDS communications, **samples** represent the data that has been received by the middleware and may be made available to the subscribing application (filters or other QoS policy settings may preclude samples from reaching the application). Each received sample is associated with a particular **instance**.

In general, samples and instances are stored in the DataReader until they are explicitly removed by the subscribing application (see Section *8.3.7 Read (or Take) Data* for details), or the CoreDX DDS middleware removes them based on various QoS policy settings. Similar to the DataWriter management, the specific rules for maintaining and deleting samples from the DataReader are different from the rules for maintaining and deleting instances. For this reason, it is possible for all samples on an instance to be removed, while the instance remains (with no associated samples). In contrast, it is *not* possible to remove an instance while any samples associated with it remain.

10.4 Instance Lifecycles

Instances are used to manage the data lifecycle. Instances are created (registered), updated (written), and deleted (disposed). For example, consider the three instances above. Instance key=12 (Scoop) may shut down for the day while the other two instances (Bob and Wendy) are still working (and updating). The publishing application can call dispose() on Scoop (instance key=12) and the remaining instances (Bob and Wendy) will still be alive. When Scoop comes online again, the publishing application can start updating that instance (which will retain the same instance handle). These data lifecycle operations are covered in detail in the following sections.

10.4.1 Registering Instances

Instances must be ***registered*** with the DataWriter before any samples associated with that instance can be written (or deleted). As a convenience, CoreDX DDS will automatically register an instance when the application calls one of the write(), unregister_instance(), or dispose() operations without first registering the instance.

Publishing applications can also explicitly register an instance by calling DataWriter::register_instance(). The register_instance() operation returns

an instance handle which can be used to improve the performance of subsequent calls to write().

The below example illustrates the use of DataWriter::register_instance() and DataWriter::write().

Example (C++)

```
HelloMessageDataWriter  dw;
HelloMessage            bobData, wendyData;
InstanceHandle_t        bobHandle, wendyHandle;
ReturnCode_t            retval;

bobData.sender_id   = 42;
bobData.sender_name = strdup("Bob the Builder");
bobData.msg         = strdup("Hello");

wendyData.sender_id = 45;
wendyData.sender_name = strdup("Wendy");
wendyData.msg = strdup("Good Morning, Bob!");

/* calling write() without an instance handle forces
 * CoreDX DDS to register this instance (key=42)
 */
retval      = dw . write( bobData, HANDLE_NIL );

/* the instance can later be 'looked up'
 */
bobHandle   = dw. lookup_instance( bobData );

/* Calling register_instance() first allows for
 * subsequent optimized calls to write()
 */
wendyHandle = dw . register_instance( wendyData );
retval      = dw . write( wendyData, wendyHandle );

delete[] wendyData.msg;
wendyData . msg = strdup( "Good night, everyone!" );

/* Changing the 'msg' in wendyData does not change the
 * key value, and therefore does not change the instance
 * or instance handle.
 */
retval      = dw . write( wendyData, wendyHandle );
```

Figure 10-1: Register Instances Example

Register instance operations are applicable only to DataWriters, and are not communicated to DataReaders. In fact, DataReaders do not have a concept of *registered* instances. Instead, a DataReader has a concept of an *Alive instance state*. Instances have an Alive state if they have at least one alive DataWriter actively writing data samples on the instance. Refer to *Section 8.4 Sample Status Information (SampleInfo)* for additional information on *instance states*. In order to manage instance states, DataReaders store a list of alive, actively writing DataWriters for each instance.

CoreDX DDS can support millions of unique instances in each DataWriter and DataReader.

10.4.2 Unregistering Instances

The publishing application can ***unregister*** previously registered instances by calling the DataWriter::unregister_instance() operation. This indicates the application will no longer be writing any samples for this instance. This is not the same as disposing the instance (which indicates the instance no longer exists). The unregister operation simply indicates that *this* DataWriter will no longer be publishing updates on the instance.

After an unregister operation, the instance handle associated with the unregistered instance is invalid. This is because the CoreDX DDS middleware may have removed all records of that instance. After an unregister operation, the instance handle may be reused for a different instance. The application may re-register the instance and then continue to publish samples or a dispose on the instance.Unregister operations are communicated to matched DatasReaders as a sample with the *valid_data* flag set to false and indicate that the DataWriter is no longer actively writing on this instance. The DataReader will remove this DataWriter from the instance's list of active DataWriters. When this list of alive, actively writing DataWriters becomes empty, the state of the instance in the DataReader Cache will change to NOT_ALIVE_NO_WRITERS. Refer to *Section 8.4 Sample Status Information (SampleInfo)* for additional information on *instance states*.

When a DataWriter is deleted by the publishing application, CoreDX DDS will automatically send an unregister command to matched DataReaders for every instance the DataWriter knows about. If the publishing application exits without deleting its DataWriter Entities, the DataReader will not receive the unregister commands. In this case, the DataReader will eventually determine the DataWriter is not alive, and remove the

DataWriter from the list of alive and actively writing DataWriters on each instance the DataReader knows about.

10.4.2.1 Relationship between DataWriter::unregister and other QoS Policies

Unregistering an instance on a DataWriter configured for Exclusive Ownership (via the Ownership QoS Policy) will cause the DataWriter to relinquish ownership of the instance. Other Exclusive Ownership DataWriters may take over ownership for the instance.

DataWriters configured with Transient Local Durability will remove the instance (and all associated samples) after all *currently alive and matched* DataReaders acknowledge receiving all samples on the instance.

DataReaders configured with auto_purge_no_writers set in their Reader Data Lifecycle may delete the instance (and all associated samples), if the DataWriter that sent this unregister command was the last active DataWriter on this instance.

10.4.3 Disposing Instances

The publishing application can ***dispose*** previously registered instances by calling the DataWriter::dispose() operation. This indicates that the instance no longer exists. This is different than the unregister operation (which indicates this DataWriter is no longer writing on this instance). CoreDX DDS treats a dispose very much like an application written data sample. The dispose command is stored by the DataWriter to be communicated to matched DataReaders. If the instance for this dispose sample is not already registered, CoreDX DDS will automatically register it before processing the dispose operation.

Dispose operations are communicated to matched DataReaders as a sample with the *valid_data* flag set to false. The DataReader will change the state of the associated instance to NOT_ALIVE_DISPOSED. Refer to *Section 8.4 Sample Status Information (SampleInfo)* for additional information on *instance states*. Note that the DataWriter will still be considered an alive, actively writing DataWriter on this instance.

10.4.3.1 Relationship between Dispose and other QoS Policies

DataReaders configured with auto_purge_disposed set in their Reader Data Lifecycle will delete the instance (and all associated samples) on receipt of a dispose command.

10.4.4 Instance Handles

An **instance handle** is a value that can be used to uniquely identify a *registered* instance. An instance handle is generated when an instance is registered (returned from a DataWriter::register_instance() call) and will be used to identify the instance until that instance is unregistered. ***The instance handle is valid only while the instance is registered***. Once an instance is unregistered, the instance handle can no longer be used to identify that instance. This is a critical detail of the CoreDX DDS middleware. These instance handles are reused, and after an unregister operation, the old instance handle may identify a *different* instance. If the unregistered instance is re-registered, a different handle may be assigned for the next 'life' of that instance.

10.5 Data Cache

DataReaders and DataWriters contain a Data Cache for storing data samples and instances. The DataWriter's Data Cache contains samples and instances that have been written, and the DataReader's Data Cache contains samples and instances that have been received. These data caches are sized and managed according to the configuration of the several QoS policies as shown in Figure Below.

Table 10-2: Instance Example

QoS Policy	Configuring this policy on the:	Will effect Cache Management on the:
RELIABILITY	DataWriter and DataReader DataReader	DataWriter DataReader
DURABILITY	DataWriter	DataWriter and DataReader
HISTORY	DataWriter DataReader	DataWriter DataReader
RESOURCE_LIMITS	DataWriter DataReader	DataWriter DataReader
READER_DATA_LIFECYCLE	DataReader	DataReader

QoS Policy	Configuring this policy on the:	Will effect Cache Management on the:
WRITER_DATA_LIFECYCLE	DataWriter	DataReader
OWNERSHIP	DataWriter	DataReader
LIFESPAN	DataWriter	DataWriter and DataReader
Filters (TIME_BASED_FILTER, content filters)	DataReader	DataReader

In general, data *samples* are added to the Data Cache as they are written (for a DataWriter) or received (for a DataReader). In general, these samples are removed when they are no longer needed by the application or the CoreDX DDS middleware. The specific management of samples in the Data Caches is described in the following sections.

In general, data *instances* are added to the Data Cache as they are registered (for a DataWriter) or received (for a DataReader). In general, these instances are removed when they are no longer needed by the application or the CoreDX DDS middleware. The management and removal of instances is different than samples, and in fact, it is possible for instances to be managed in the DataWriter and DataReader Data Caches even when there are no samples associated with the instance in the DataCache. The specific management of instances in the Data Caches is described in the following sections.

10.5.1 DataWriter Cache

The DataWriter's Data Cache contains samples that have been written and instances that have been registered (or written, since calling DataWriter::write() automatically registers an instance).

10.5.1.1 Samples in the DataWriter Cache

Samples are added to the Data Cache as they are written by the application. Samples will be stored n the DataWriter Cache until they are initially written, and may be stored longer to meet Reliability and Durability QoS settings. Samples can be removed from the cache by the application (calling

DataWriter::unregister()) or by the CoreDX DDS infrastructure, based on QoS policy settings. Samples are removed from the DataWriter cache under the following conditions:

1. The CoreDX DDS middleware in the publishing application has completed writing the sample. This happens when:
    a. The CoreDX DDS middleware writes the sample to all *Best Effort* DataReaders AND
    b. (Only if the DataWriter is *Reliable* and *Volatile*) The CoreDX DDS middleware has received an acknowledgement from all *Reliable* DataReaders
2. OR: Samples expire based on the Lifespan duration
3. OR: A DataWriter has a History QoS Policy *kind* of *KEEP_LAST* and the cache already holds History *depth* samples and a new sample is created by write(), unregister() or dispose()
4. OR: A Best Effort DataWriter has non-INFINATE *max_samples* or *max_samples_per_instance* Resource Limits and the cache already holds the maximum samples and a new samples is created by write(), unregister() or dispose().

It is important to note that some combinations of QoS settings will cause the DataWriter cache to grow infinitely, consuming more and more run time memory. This can happen with a Durability kind of TRANSIENT_LOCAL and Reliability kind of RELIABLE, a History kind of KEEP_ALL, and Resource Limits set to infinite. With this combination of QoS settings, the application must manage instances to meet application or machine resource limitations.

10.5.1.2 Instances in the DataWriter Cache

Instances are added to the DataWriter Cache when the publishing application registers an instance that is not already registered. Every sample belongs to an instance, and the instance must be registered before a sample on that instance samples can be created. The application can explicitly register an instance by calling DataWriter::register_instance(), or CoreDX DDS will automatically register the instance when the application attempts to create a sample without first registering its associated instance.

Instances are removed from the DataWriter Cache when the publishing application unregisters an instance. This must be done explicitly by calling DataWriter::unregister_instance().

10.5.2 DataReader Cache

The DataReader's Data Cache contains samples and instances that have been received (subject to filters). These samples and instances may or may not have been already read (seen) by the application.

10.5.2.1 Samples in the DataReader Cache

Samples are added to the DataReader Cache when they are received by the DataReader and the sample passes any filters configured on the DataReader and there is room in the DataReader Cache for the new sample. If there is not room in the DataReader Cache, the new sample will be added only if the History QoS policy is configured to *KEEP_LAST.*

Samples are eligible to be removed from the DataReader cache under the following conditions:

1. When the application uses DataReader::take().
2. If the Lifespan QoS is used, samples will be removed from the data cache after they are expired.

3. If the Data Cache is full and the History kind is KEEP_LAST, the oldest sample will be removed to make room for a newly received sample.
4. When a Best Effort DataReader has non-INFINITE max_samples or max_samples_per_instance Resource Limits and the cache already holds the maximum samples and a new sample is received.
5. When a DataReader has non-INFINITE *autopurge_nowriter_samples_delay* or *autopurge_disposed_samples_delay* and an instance state is determined to be NOT_ALIVE_NO_WRITERS or NOT_ALIVE_DISPOSED; (associated samples will be removed after the specified delay)

10.5.2.2 Instances in the DataReader Cache

Instances are added to the DataReader Cache when a sample belonging to the instance is received by the DataReader and the instance does not already exist in the DataReader Cache. The instance is created even if the associated sample is *not* added to the DataReader Cache. Samples may not be added to the DataReader Cache because of filters applied on the DataReader or because of the configuration of the History and Resource Limits QoS policies.

Instances are eligible to be removed from the DataReader Cache when an instance has a state of NOT_ALIVE_NO_WRITERS and there are no associated samples.

Chapter 11 Application Data Types

11.1 Overview

Every DDS Topic contains one and only one ***Data Type***, a user defined data type that is used when communicating on the Topic. In most cases, the application developer defines these DDS Data Type(s) in the Data Definition Language (DDL) format. A compiler is used to translate these DDL type definitions into the appropriate programming language for inclusion into an application.

CoreDX DDS also supports *dynamic types,* which are Data Types that are not defined at compile type. With dynamic types, it is possible to publish and subscribe to a discovered Topic with a discovered Data Type. In this scenario, the application has no knowledge of the data type until the Topic is discovered at run time. A complete discussion of dynamic types can be found in Chapter 18: Dynamic Types.

11.2 Why Define the Data Types?

CoreDX DDS is data-centric. This means that the structure and contents of application data is known and used by the CoreDX DDS middleware. This allows the CoreDX DDS middleware to perform advanced data management operations that are not available in other message oriented middleware technologies. For example, instance and sample history are enabled by identifying 'key' fields in the data to identify unique data instances. This can be compared to key fields in relational database technologies. Each key uniquely identifies a collection of related records. In DDS, the key is used to identify a data 'instance'. Updates to a data instance are referred to as 'samples'. The CoreDX DDS middleware can maintain historical samples for each instance (see the HISTORY Quality of Service). Furthermore, the CoreDX DDS middleware can apply a Content Filter to data samples. Content filters are comparable to an SQL query that selects a sub-set of available data based on conditions. These are a few examples of the power that a data-centric middleware can offer.

Furthermore, because the data types are well known, the DDS API and compilers can enforce the usage of proper types throughout the application code. This can avoid potentially difficult bugs related to subtle mismatches of data throughout the distributed system.

The benefits of data-centric middleware are possible because the middleware has an understanding of the data structures used in your application. Part of the DDS development workflow includes defining the application data types and registering them with the CoreDX DDS middleware.

11.3 Data Types and Discovery

Data Types play an important role in the process of DDS dynamic *discovery*. (Additional information on CoreDX DDS discovery can be found in Part 4:Chapter 16 *CoreDX DDS Discovery*). A DataWriter will match with a DataReader only if the data type published by the DataWriter matches the data type subscribed to by the DataReader. This provides additional type safety for DDS applications because a DataReader cannot receive data in an unexpected format (a commonly occurring programming error when using other communication middleware products).

11.4 Data Normalization

The process of DDS Data Architecture is very similar to that of designing a Relational Database schema. Because the data structures used by your application will be conveyed between applications, and possibly over a network, it is important that they be designed with efficiency in mind. This 'data normalization' process is important to effective deployment of DDS technology.

11.5 Data Type Definition

Once the appropriate data structures have been designed, they must be written in a language that the CoreDX DDS middleware can understand. This is called the Data Definition Language (DDL) and is a subset of the Interface Definition Language (IDL) found in the CORBA standards.

DDL is a simple and flexible language for defining data types. It has a rich set of primitive and complex data types, and there are defined mappings from IDL to many common programming languages. This makes DDL a good language for DDS.

11.6 DDL Syntax

A DDS Data Type is always a ***structure***, which may contain any combination of basic and constructed types, including embedded structures. Table 11-1is a list of basic types supported in the CoreDX DDS DDL.

Table 11-1: Basic User Defined Types

Basic Type	Description
short	2 bytes
long	4 bytes
long long	8 bytes
unsigned short	2 bytes
unsigned long	4 bytes
unsigned long long	8 bytes
float	4 bytes
double	8 bytes
long double	16 bytes
char	1 byte
boolean	1 byte
octet	binary data, 1 byte
enum	Enumerated type, 4 bytes
string	bounded and unbounded
constant	constant type, always a number

Table 11-2 is a list of constructed types supported in the CoreDX DDS DDL.

Table 11-2: Constructed User Defined Types

Constructed Type	Description
struct	Structure type
union	Union type
array	Single or multi dimensional
sequence	bounded and unbounded

CoreDX DDS DDL does not support the ***any*** type.

11.7 DDL Language Mappings

The CoreDX DDL compiler generates source code for use in application programs. These files contain, among other things, a translation of the IDL specified data type into a language specific data type. The mapping of IDL to programming language follows the standards of the OMG, with a few deviations focused on performance related to the DDS API.

The following subsections identify how the various IDL data types are mapped to each of the supported CoreDX DDS programming languages.

11.7.1 Basic Types

Table 11-3: Primitive Data Type Mapping

IDL Type	C	C++	C#	C++
char	char	char	char	char
octet	unsigned char	unsigned char	byte	byte
short	short	short	short	short
unsigned short	unsigned short	unsigned short	ushort	short

IDL Type	C	C++	C#	C++
long	int	int	int	int
unsigned long	unsigned int	unsigned int	uint	int
long long	int64_t	int64_t	long	long
unsigned long long	unsigned int64_t	unsigned int64_t	ulong	long
float	float	float	float	float
double	double	double	double	double
long double	N/A	N/A	N/A	N/A
string	char *	char *	String	String

11.7.2 Complex Types

Table 11-4: Complex Type Mapping

IDL Type	C	C++	C#	Java
struct	struct	struct	class	class
union	struct	class	class	class
array	[]	[]	[]	[]
sequence	struct	vector	[]	[]
enum	#define	enum	enum	class

11.7.2.1 Unions

The mapping of Union types into a target programming language is similar to a standard structure with some added features. First, the structure includes a *discriminator* to identify which of the possible union types are present. The discriminator is named _d in the 'C' language mapping, and in C++ and Java, methods to access and set the discriminator are provided. Further, in C++ and Java, accessors and setters for each of the union alternatives are provided.

11.7.2.2 Sequences

The sequence data type is used in a few places throughout the DDS API, in addition to an available construct for user defined data types. It is most prevalent in the DataReader API where the sequence is used to return a collection of samples and SampleInfo items back to the caller.

Sequences are mapped differently for each target language in the interest of optimizing performance.

11.7.2.2.1 C Implementation of Sequences

In the C language binding, sequences are mapped to a CoreDX DDS specific data structure that provides semantics similar to the vector container in C++.

A C sequence data structure is described in Table 11-5. CoreDX DDS applications using the C language binding can access these fields directly, or use the sequence helper functions defined in Table 11-6.

Table 11-5: Sequence Data Structure

Sequence Field Name	Type	Description
_buffer	Array of the sequence type	This field contains the sequence data.
_maximum	The sequence to initialize	For future use: This value is not used by CoreDX DDS.
_length	(none)	The number of elements in the sequence
_size	(none)	The size of the sequence; the sequence has enough memory

Sequence Field Name	Type	Description
		allocated to hold this many elements. This does not indicate the number of elements in the sequence.
_own	(none)	For future use: This value is not used by CoreDX DDS.
_item_size	(none)	The size of 1 sequence element (determined by the element type).

CoreDX DDS provides the following C functions to interact with sequences.

Table 11-6: C Sequence Functions

Function	Arguments	Return	Description
DECLARE_SEQ()	Sequence type Sequence name	(none)	This is a MACRO that will declare a C sequence data structure.
INIT_SEQ()	sequence	(none)	This is a MACRO that will initialize a C sequence data structure. A C sequence MUST be initialized before any of the following functions are used.
seq_add()	Pointer to a sequence Pointer to a sequence element	(none)	Add an element to the end of the sequence. This will increase _length by 1, and if necessary will allocate additional memory and increase _size.
seq_at()	Pointer to a sequence index	sequence element	This is a MACRO, it returns the element at the specified index.

Function	Arguments	Return	Description
seq_clear()	Pointer to a sequence	(none)	Remove all elements in the sequence (it will not free the elements), and set the _size and _length to 0.
seq_copy()	Pointer to the TO sequence Pointer to the FROM sequence	(none)	Copy the elements from the FROM sequence to the TO sequence. The original TO sequence will be destroyed.
seq_free()	Pointer to a sequence	(none)	Removes all elements in the sequence (it will not free the elements) and deletes the sequence memory.
seq_get_length()	Pointer to a sequence	unsigned int	Return the value of _length.
seq_get_maximum()	Pointer to a sequence	unsigned int	Return the value of _maximum.
seq_get_own()	Pointer to a sequence	unsigned char	Return the value of _own. Not currently used by CoreDX DDS.
seq_get_size()	Pointer to a sequence	unsigned int	Return the value of _size.
seq_set_length()	Pointer to a sequence New length (unsigned int)	unsigned int	Set the value of _length.
seq_set_maximum()	Pointer to a sequence New maximum (unsigned int)	unsigned int	Set the value of _maximum.
seq_set_own()	Pointer to a sequence Own value (unsigned char)	unsigned char	Set the value of _own. Not currently used by CoreDX DDS.

Function	Arguments	Return	Description
seq_set_size()	Pointer to a sequence New size (unsigned int)	unsigned int	If the current value of _length is greater than the new size, the sequence is truncated to the new size. Set the value of _size, and if necessary, allocate additional memory so the sequence can hold at least new size elements.

11.7.2.2.2 C++ Implementation of Sequences

In the C++ language binding, the mapping is to a template implementation of a sequence that provides semantics similar to the C sequence implementation, and an API similar to the Standard Template Library (STL) vector class.

Table 11-7: C++ Sequence Methods

Function	Arguments	Return	Description
sequence()	(none)	New sequence	Default constructor: construct an empty sequence
sequence(const sequence &x)	sequence to copy	New sequence	Copy constructor: copy the contents of a sequence
~sequence()	(none)	(none)	Destructor: destroy the sequence and release resources, this method destroys the sequence contents.
Operator=(const sequence &x)	sequence to assign	sequence	Assignment operator: copies sequence contents.
size()	(none)	number of elements (unsigned int)	Returns the current size (number of elements) of the sequence.

Function	Arguments	Return	Description
capacity()	(none)	capacity of sequence (unsigned int)	Returns the current capacity (number of elements) of the sequence. The equation "size <= capacity" is always true.
reserve(uint32_t n)	new capacity (unsigned int)	(none)	Set the capacity of the sequence. If n is larger than current capacity, then the capacity is increased to 'n'. If 'n' is smaller than current capacity, then the capacity is reduced to 'n'. If capacity is reduced below 'size', then 'size' will be reduced to match 'capacity'. In the case of a reduction in 'size', the eliminated elements will be destroyed.
resize(uint32_t n)	new size (unsigned int)	(none)	Set the capacity and the size of the sequence. If n is larger than current capacity, then the capacity is increased to 'n'. If 'n' is smaller than current capacity, then the capacity is reduced to 'n'. If capacity is reduced below 'size', then 'size' will be reduced to match 'capacity'. In the case of a reduction in 'size', the eliminated elements will be destroyed.
empty()	(none)	bool	Test if the sequence is empty.
shrink_to_fit()	(none)	(none)	Reduce 'capacity' to match 'size'.

Function	Arguments	Return	Description
operator[]	position (unsigned int)	sequence element	Set the value of _length.
operator[] const	position (unsigned int)	sequence element	Set the value of _length.
at[]	position (unsigned int)	sequence element	Set the value of _maximum.
front()	(none)	sequence element	Access the first element in the sequence.
back()	(none)	sequence element	Access the last element in the sequence.
push_back()	(none)	(none)	Add an element after the last, increment size and (if required) capacity.
pop_back()	(none)	(none)	Reduce 'size' by 1.
swap(sequence &x)	another sequence to swap with	(none)	Swap the contents of this sequence with the contents of another sequence.
clear()	(none)	(none)	Clear the contents of this sequence.

11.7.2.2.3 Java Implementation of Sequences

In the Java language binding, sequences are mapped to a simple Array type which is comparable to the std::vector class.

11.7.2.2.4 C# Implementation of Sequences

In the C# language binding, sequences are mapped to a List type.

11.8 Creating a DDL File

The syntax of DDL is flexible and is similar to other languages such as C. The DDL file can contain any combination of:

- C/C++ style comments
- C compiler directives
  #if, #ifdef, #else, #endif,
  #include, etc.
- namespace / module
- constant definitions
- enumerated type definitions
- typedefs
- structures (these become DDS data types)
- unions

Figure 11-1: Example DDL file provides an example of a DDL file.

Example DDL file

```
//=========================================================
// CoreDX DDS DDL example
//=========================================================

struct SenderType
{
  string firstname;
  string lastname;
};

struct StringMsg
{
   SenderType       sender;
   long             time_sent;
   sequence<string> old_msgs;
   string           msg;
};
```

Figure 11-1: Example DDL file

The syntax for these DDL file elements adheres to the OMG's IDL syntax specification as defined in Section 7 of the OMG's Common Object Request Broker Architecture (CORBA) Specification, Version 3.2, Part 1: CORBA Interfaces. In particular, Section 7.11 describes IDL type definitions, which are the primary focus of DDL files. The CoreDX DDS DDL compiler (coredx_ddl) can parse fully compliant IDL files; it will use the type definitions and ignore any interface definitions when generating CoreDX DDS code.

11.9 Specifying Keys

The application developer, when creating a DDS data type, can specify one or more attributes of the DDS data type as a ***key***. The CoreDX DDS middleware uses those key attributes to organize the data into ***instances*** (see the *Instances and Samples* chapter for additional information).

A key can be any field in the DDS Data Type, except for *sequences, unions,* and *enums*. The application developer may define any number of fields to be a key field. All the fields labeled as a key field are concatenated together to form one key for the DDS Data Type.

To specify a field to be a key, add the string "__dds_key" in front of the field definition. The CoreDX DDS DDL compiler defines a compiler flag: "DDS_IDL", and it is common to use this to provide compatible IDL in the DDL files. The following DDL provides several examples of using keys in the DDL.

Example DDL file

```
//=====================================================
// CoreDX DDS DDL example - using keys
//=====================================================

// This #ifdef block allows the following DDL to be
// compatible with IDL

#ifdef DDS_IDL
#define DDS_KEY __dds_key
#else
#define DDS_KEY
#endif

// the 'sender_id' field is defined as a key
struct StringMsg1
{
```

```
   struct SenderType  {
      string first_name;
      string last_name;
   }                          sender;
   DDS_KEY long               sender_id;
   long                       time_sent;
   sequence<string>           old_msgs;
   string                     msg;
};

// the 'sender' field is defined as a key
struct StringMsg1
{
   DDS_KEY struct SenderType  {
      string first_name;
      string last_name;
   }                          sender;
   long                       sender_id;
   long                       time_sent;
   sequence<string>           old_msgs;
   string                     msg;
};

// the key is a combination of the sender's last name
// and the sender_id
struct StringMsg1
{
   struct SenderType  {
      string first_name;
      DDS_KEY string last_name;
   }                          sender;
   DDS_KEY long               sender_id;
   long                       time_sent;
   sequence<string>           old_msgs;
   string                     msg;
};
```

Figure 11-2: DDL keys example

If a user defined structure with a key is embedded inside of another structure, the container structure inherits the key fields of the original structure.

11.10 Using the DDL Compiler

The CoreDX DDS DDL compiler (coredx_ddl or coredx_ddl.exe) compiles the DDS types defined in the DDL file and generates type-specific source code to be compiled and linked with a CoreDX DDS application. The CoreDX DDS DDL compiler provides some command line arguments to tailor the behavior.

Table 11-8: coredx_ddl command line options

coredx_ddl option	Default	Description
-h	n/a	Help: print coredx_ddl usage information
-f *filename*	No default, must be specified	File: provide the DDL file to process
-p *preprocessor*	cpp (linux) cl.exe /E (win)	Preprocessor: provide the preprocessor to use. The location of the specified preprocessor must be in the PATH.
-l *language*	c	Language: one of "c", "cpp", "csharp", "java" provide the language to use for the generated source code.
-d *directory name*	Current directory	Output Directory: provide an alternate directory to put the resulting generated code. The default is the current working directory.
-b *name*	DDL filename	Basename: provide an alternate filename prefix to use for the generated files. The default is the basename of the DDL filename. For example, a DDL file named "hello.ddl" will, by default, produce generated files named like "hello.h", "helloTypeSupport.h", etc. The user can provide this argument to change the generated filename prefix to a string specified by *name*. -b option only valid for C and C++ languages

coredx_ddl option	Default	Description
-e *endian*	HOST CPU architecture	Endian: one of "b" or "l", provide the byte order (**b**ig endian or **l**ittle endian) to use when marshalling published data to transmit over the wire. The default is the endian of the HOST platform (the platform where the coredx_ddl compiler is running). This should be used when the TARGET platform has a different endianness than the HOST platform. If the endianness of the TARGET platform is not set correctly, DDS applications that perform a read or take operation will have undefined behavior and may segfault.
-D *preprocessor symbol*	n/a	This option is used to specify pre-processor defines. Predefined by CoreDX DDS: *DDS_IDL*, *COREDX_DDS*
-I *include path*	empty	This option provides a path that will be searched to satisfy '#include' directives found in the DDL file(s).
-a *alignment flag*	0	0 == don't count CDR ENCAP HDR in alignment. This is the default in CoreDX DDS versions 3.5.3 and newer, and is interoperable with other DDS implementations. 1 == count CDR_ENCAP_HDR is alignment. This value is interoperable with older CoreDX DDS versions (before v3.5.3).

The CoreDX DDS DDL compiler generates several source code files. All generated files are written to the current working directory (the directory from which coredx_ddl was run).

For additional CoreDX DDL Compiler options for use with Dynamic Types, refer to Part 4:Chapter 18: Dynamic Types.

11.11 Generated Code

The CoreDX DDS DDL compiler will generate several source files, listed in Table 11-9. Detailed descriptions of the generated files are listed below.

Table 11-9: Generated source code file names

C filenames	C++ filenames	C# filenames	Java filenames
name.h **name.c**	name.hh name.cc	name.cs	name.java
nameTypeSupport.h **nameTypeSupport.c**	nameTypeSupport.hh nameTypeSupport.cc	nameTypeSupport.cs	nameTypeSupport.java
nameDataReader.h **nameDataReader.c**	nameDataReader.hh nameDataReader.cc	nameDataReader.cs	nameDataReader.java
nameDataWriter.h **nameDataWriter.c**	nameDataWriter.hh nameDataWriter.cc	nameDataWriter.cs	nameDataWriter.java

11.11.1 Type Definition

The type definition files (name.h, name.c in C) contain the language specific data type declarations for the DDS data types defined in the DDL file. They also include basic initialization, copy, and delete operations for the data types.

11.11.2 Typed TypeSupport Definition

The type support files (nameTypeSupport.h, nameTypeSupport.c in C) contain the language and type specific TypeSupport structures. For languages that support heredity, these are classes that derive from the base TypeSupport class.

11.11.3 Typed DataReader and DataWriter Definitions

The data reader (nameDataReader.h, nameDataReader.c) and data writer files (nameDataWriter.h, nameDataWriter.c) contain the language and type specific DataReader and DataWriter structures. For languages that support heredity, these are classes that derive from the base DataReader and DataWriter classes.

The application should use the type specific operations for DataReader and DataWriter calls.

Chapter 12 Quality of Service Features

One of the powerful features of DDS is the support for various ***Quality of Service*** (QoS) settings. QoS settings allow the application developer to tailor the behavior of publishers, subscribers, and the communications between them.

Most DDS Entities, from a DomainParticipantFactory down to the DataReader and DataWriter, have a set of QoS settings that apply. The QoS settings are contained in a structure. For example, a DomainParticipantFactory has a DomainParticipantFactoryQos structure containing all the applicable QoS settings.

The QoS settings for an entity can be established when the entity is created, or by using the get_qos() and set_qos() methods on each entity, as the following C example illustrates.

Sample C Application Code

```
DDS_Subscriber         subscriber;
DDS_DataReader         dr;
DDS_DataReaderQos      drqos;
DDS_DataReaderListener dr_listener;

/* … code deleted … */

/* Setup a non-default DataReader QoS structure */

DDS_Subscriber_get_default_datareader_qos(subscriber, &drqos);
drqos.history.kind  = DDS_KEEP_LAST_HISTORY_QOS;
drqos.history.depth = 5;

/* EXAMPLE 1: Define the DataReader QoS at creation */

dr = DDS_Subscriber_create_datareader(subscriber,
                                      topic_descr,
                                      &drqos, &dr_listener,
                                      DDS_DATA_AVAILABLE_STATUS);

/* EXAMPLE 2: Set the QoS after creating the DataReader */
DDS_DataReader_set_qos(dr, &drqos);
```

Figure 12-1: Configuring QoS - Sample C Code

12.1 QoS Compatibility

Many QoS settings are applicable to more than one entity. And in order for effective communications, in some cases the QoS setting must be compatible between publishing entities and subscribing entities. Publishers and DataWriters ***offer*** a QoS configuration. Subscribers and DataReaders ***request*** a QoS configuration. If the Publisher and DataWriter offer a configuration setting that is ***at least*** what the Subscriber and DataReader requested, this is considered a match, even if the QoS configurations are not the same.

For example, the DURABILITY QoS setting indicates weather a publishing application will save previously published data and whether a subscribing application expects to receive data that was published before it was created. Consider a subscribing application requesting a DURABILITY QoS to be able to receive history (data published before the DataReader existed), and a publishing application offering a DURABILITY QoS indicating it will not save any data after it has been published. It is impossible for this publishing application to meet the request of this subscribing application, and effective communication will not occur. However, if the subscribing application requests a DURABILITY QoS to not receive any history, and a publishing application offers a DURABILITY QoS to make history available, the QoS settings match. In this case, the publishing application is offering more than the subscribing application is requesting.

When attempting to match up publishing applications with subscribing applications, the CoreDX DDS middleware will consider the QoS settings on both sides (as well as on the Topic). If all QoS settings are compatible, the publishing application and subscribing application will be "matched". If any QoS settings are incompatible both the publishing and subscribing applications are notified. The OfferedIncompatibleQos status is updated on the publishing application and the RequestedIncompatibleQos status is updated on the subscribing application. For information on how to retrieve communication statuses, see the *Communication Status* chapter.

Table 12-1 lists the compatibility of each QoS setting (whether or not the QoS setting must be compatible between publishing entities, subscribing entities, and topics).

12.2 QoS Mutability

Many QoS settings can be changed only before the DDS Entity is *enabled*. However, there are some QoS settings that can changed dynamically at any time. These QoS settings are considered *mutable* or changeable. Table 12-1 lists the mutability of each QoS setting (whether or not the QoS setting can be dynamically changed at any time). Attempting to change a QoS setting that is not *mutable* using a set_qos() operation will return DDS::RETCODE_IMMUTABLE_POLICY.

12.3 Quality of Service Details

Table 12-1 lists all the standard DDS QoS policies, along with compatibility and mutability characteristics. Following the table is a detailed list of all supported QoS features from the OMG DDS specifications and description of their use. CoreDX DDS supports additional policies beyond those defined in the OMG DDS specifications. These additional QoS policies are described in Part 4: CoreDX DDS Extensions.

Table 12-1: QoS Summary

QoS Setting	Must be compatible	Dynamically Changeable
DEADLINE	YES	YES
DESTINATION_ORDER	YES	no
DURABILITY	YES	no
DURABILITY_SERVICE	no	no
ENTITY_FACTORY	no	YES
GROUP_DATA	no	YES
HISTORY	no	no
LATENCY_BUDGET	YES	YES
LIFESPAN	(n/a)	YES

QoS Setting	Must be compatible	Dynamically Changeable
LIVELINESS	YES	no
OWNERSHIP	YES	no
OWNERSHIP_STRENGTH	(n/a)	YES
PARTITION	no	YES
PRESENTATION	YES	no
READER_DATA_LIFECYCLE	(n/a)	YES
RELIABILITY	YES	no
RESOURCE_LIMITS	no	no
TIME_BASED_FILTER	(n/a)	YES
TOPIC_DATA	no	YES
TRANSPORT_PRIORITY	(n/a)	YES
USER_DATA	no	YES
WRITER_DATA_LIFECYCLE	(n/a)	YES

12.3.1 DEADLINE

The ***Deadline*** QoS policy is used when a Topic is expected to have each instance updated periodically. The DataWriter offers a *Deadline* contract where the application guarantees to update each instance every *n* time period. The DataReader requests a *Deadline* contract where the application expects each instance to be updated every *n* time period.

When a writing application does not satisfy the DataWriter's Deadline period (configured in its QoS policy), the offered_deadline_missed status is updated. The writing application may choose to be notified of this event through any of the offered notification methods (refer to *Communication Status* for more information).

When a writing application does not satisfy a matched DataReader's Deadline period (configured in it's QoS policy), the requested_deadline_missed status is updated. The reading application may choose to be notified of this event through any of the available notification methods (refer to *Communication Status* for more information).

This is a QoS policy that must be compatible before DataReaders and DataWriters will match. The DataWriter must have a *Deadline* <= the DataReader's *Deadline* before the DataWriter and DataReader will communicate. If the *Deadlines* are not compatible, CoreDX DDS will generate an IncompatibleQos status (see the *Communication Status* chapter for additional information).

12.3.2 DESTINATION_ORDER

The ***Destination Order*** policy determines the logical order at reception time of data samples for an instance. This is important when the infrastructure must determine which samples to keep at the DataReader, based on other QoS policies like HISTORY and RESOURCE_LIMITS. The possible values for ***Destination Order*** are *by reception timestamp* and *by source timestamp*. When set to *by reception timestamp,* CoreDX DDS will use the reception time to determine the order of samples. When set to *by source timestamp,* CoreDX DDS will use the timestamp set by the publisher to determine the order of samples, regardless of when the data sample was received.

12.3.3 DURABILITY

The ***Durability*** policy controls whether or not CoreDX DDS will make already published data available to late joining DataReaders. The publish-subscribe paradigm offered by CoreDX DDS allows applications to write data even when there are no current readers on the network. Further, a DataReader has the option to receive historical data (data published before this DataReader came online) in addition to currently published data. The ***Durability*** policy allows this configuration.

The possible values for ***Durability*** are *Volatile*, *Transient Local*, *Transient*, and *Persistent*. When set to *Volatile*, CoreDX DDS will not save previously published data for late joining readers. When set to *Transient Local*, CoreDX DDS will save previously published data for the lifespan of the DataWriter for late joining readers. With a *Transient Local* setting, once the DataWriter is destroyed, its published history data is no longer available. When set to *Transient*, CoreDX DDS will save previously published data for the lifespan of the publishing application for late joining readers. With a *Transient* setting,

once the publishing application exits, its published history data is no longer available. When set to *Persistent*, CoreDX DDS will save previously published data in permanent storage, where it can outlive the publishing application and a system reset. This history data is available to late joining readers.

The 'wait_for_historical_data' may be used on DataReaders with a ***Durability*** setting of *Transient Local* or stronger to block the application until all historical data has been received by the DataReader. Refer to section 8.5.2 *Wait for Historical Data* for additional information.

This is a QoS policy that must be compatible before DataReaders and DataWriters will match. The DataWriter must have a ***Durability*** <= the DataReader's ***Durability*** before the DataWriter and DataReader will communicate, where the ***Durability*** values are ordered so that *Volatile* < *Transient Local* < *Transient* < *Persistent*. If the ***Durabilities*** are not compatible, CoreDX DDS will generate an IncompatibleQos status (see the *Communication Status* chapter for additional information).

The number of data samples and instances stored for any ***Durability*** setting is determined in part by the configuration of the ***History*** and ***Resources Limits*** QoS policies.

CoreDX DDS currently supports only the *volatile* and *transient local* values for ***Durability***.

12.3.4 DURABILITY_SERVICE

The ***Durability Service*** QoS policy is applicable only when the ***Durability*** QoS policy is set to *Transient* or *Persistent*. This policy configures the duration and amount of data stored.

CoreDX DDS currently does not support the ***Durability Service*** QoS policy.

12.3.5 ENTITY_FACTORY

The ***Entity Factory*** QoS policy controls the behavior of create_xxx() and delete_xxx() operations on a factory entity. Factory entities include: *DomainParticipantFactory*, *DomainPartition*, *Publisher*, and *Subscriber*.

When set to *TRUE*, all *Entities* returned by create_xxx() operations are already enabled. When set to *FALSE*, the application must explicitly call *enable()* on all created *Entities*.

The default setting for the ***Entity Factory*** QoS policy is *TRUE*.

12.3.6 GROUP_DATA

The ***Group Data*** QoS policy allows the application to attach additional information to created *Entity* objects. This data is not used by CoreDX DDS, and is made available to the application by the ***Built-in Topics***, along with other discovery information. For more information, see the 9.2 *Built-In Topics* section.

12.3.7 HISTORY

The ***History*** QoS policy (along with the ***Resource Limits*** QoS policy) controls the size and behavior of the DataReader and DataWriter data caches. The data caches may be used to buffer written data on the DataWriter, and received data on the DataReader. The ***History*** QoS policy determines how CoreDX DDS will save data samples, and the number of samples that may be saved, for each instance. The ***History*** QoS policy can provide some amount of buffering on both the publishing and subscribing sides. When combined with the ***Durability*** QoS policy on a DataWriter, this QoS policy will determine the amount of data history saved for late joining readers. On a DataReader, this policy will determine the number of samples available to return on a read() or take() operation.

The possible values for the ***History*** kind are *KEEP ALL* and *KEEP LAST*. When the ***History*** kind is configured to *KEEP_LAST*, CoreDX DDS may delete stored data samples to make room for newly written (or received) samples. When the ***History*** kind is configured to *KEEP_LAST*, CoreDX DDS will never 'bump' a stored data sample to make room for a newly written (or received) sample. When set to *KEEP LAST*, the application can define a *depth* (number of samples to keep).

A ***History*** kind of *KEEP_ALL* may be used in combination with the ***Resource Limits*** QoS policy in order to bound the number of samples and/or instances stored by CoreDX DDS.

Here is an example of how the ***History*** kinds can behave. Let us consider a scenario where a Reliable DataWriter is writing samples faster than at least one if it's matched Reliable DataReaders can read or process them. In this case, CoreDX DDS will attempt to buffer the unread samples. Samples will be buffered first at the DataReader. With a ***History*** kind of KEEP_LAST, samples are buffered up to the ***History*** depth, and then they may be overwritten. With a ***History*** kind of KEEP_ALL, samples are buffered up to

the configured ***Resource Limits.*** If ***Resource Limits*** are configured to be infinite, samples will be buffered infinitely. If ***Resource Limits*** are configured to be finite value(s) and the DataReader has buffered that configured number of samples, the DataReader will drop (and not acknowledge) any more received samples. At this point, samples will be buffered at the DataWriter.

Note that using a ***History*** set to KEEP_ALL in combination with a ***Durability*** set to TRANSIENT_LOCAL (or higher) can be a dangerous combination. The CoreDX DDS infrastructure will keep every sample ever written by the DataWriter, until the publishing application specifically *disposes* or *unregisters* the Instance (see the *Instances and Samples* chapter). This can quickly utilize all available resources for the publishing application, or the host machine if ***Resource Limits*** are not specified.

12.3.8 LATENCY_BUDGET

The ***Latency Budget*** QoS policy specifies a delay is acceptable in the time between when a publishing application writes data to when a subscribing application is notified the data is available. CoreDX DDS uses this policy as a hint – not a contract that must be monitored or enforced. By default, the ***Latency Budget*** is set to zero (0), indicating the delay should be minimized.

It may not be obvious why an application would want to configure a ***Latency Budget*** greater than zero. Here are two examples of when it may be appropriate to configure a ***Latency Budget***. First, consider a publishing application that is publishing a very high rate of data samples. If the ***Latency Budget*** is set to zero, CoreDX DDS will attempt to write every data sample onto the network as soon as it is available from the application, which may not be very efficient. In this case, setting a ***Latency Budget*** greater than zero allows CoreDX DDS to queue a handful of data samples to write in a batch, which will reduce the amount over overhead required, and may improve performance. For another example, consider a subscribing application that is receiving a very high rate of data samples. If the ***Latency Budget*** is set to zero, CoreDX DDS will notify the application for every data sample that arrives at the DataReader. However, if the ***Latency Budget*** is set to a value greater than zero, CoreDX DDS can queue received data samples, and send 1 notification for multiple available samples. In this case, the subscribing application is issuing fewer calls to read() or take() but receiving all the same data samples.

12.3.9 LIFESPAN

The ***Lifespan*** QoS policy allows CoreDX DDS to expire old data samples. The application configures an expiration duration time on a DataWriter.

A DataReader receiving data from this DataWriter will periodically check all the samples that have been received, and if any samples have *expired*, they will be removed from the DataReader cache.

A DataWriter will also periodically check all the samples in its DataWriter Cache and may remove any samples that have expired (actual removal may be delayed due to Reliability QoS policy settings).

By default, the expiration duration is 0 (meaning an infinite duration, or no expiration).

12.3.10 LIVELINESS

The ***Liveliness*** QoS policy controls the mechanism used to ensure *DataWriters* on the network remain "alive" to their matched DataReaders. The possible values for the ***Liveliness*** QoS policy are *Automatic*, *Manual by Participant*, and *Manual by Topic*.

The *Automatic* setting configures CoreDX DDS ensure all *DataWriters* within a *DomainParticipant* stay alive, without requiring any specific action from the publishing application.

The manual settings: *Manual by Participant* and *Manual by Topic* require the publishing application to periodically assert the liveliness to indicate the corresponding *Entity* is still alive. This can be explicit by calling the *assert_liveliness()* operation or implicit by writing data. The *Manual by Participant* configuration allows any one DataWriter to assert liveliness for all DataWriters within that DomainParticipant. The *Manual by Topic* configuration requires each DataWriter to assert its own liveliness.

The ***Liveliness*** QoS policy includes a *lease duration*. For a DataWriter, the *lease duration* is an offered contract that the writer will assert liveliness at least once every specified duration. For a DataReader, the *lease duration* is a request that the writer assert liveliness at least once every specified duration interval.

When a writing application does not satisfy the DataWriter's ***Liveliness*** lease duration, the liveliness_lost status is updated. The writing application may

choose to be notified of this event through any of the offered notification methods (refer to *Communication Status* for more information).

When a writing application does not satisfy a matched DataReader's liveliness period, the liveliness_changed status is updated. The reading application may choose to be notified of this event through any of the available notification methods (refer to *Communication Status* for more information).

This is a QoS policy that must be compatible before DataReaders and DataWriters will match. When configured to one of the Manual kinds, the DataWriter must have a ***Liveliness*** *lease duration* >= the DataReader's ***Liveliness*** *lease duration* before the DataWriter and DataReader will communicate. If the ***Liveliness*** is not compatible, CoreDX DDS will generate an IncompatibleQos status (see the *Communication Status* chapter for additional information).

12.3.11 OWNERSHIP

The ***Ownership*** QoS policy controls whether CoreDX DDS will allow multiple *DataWriters* to update the same instance. The possible values for ***Ownership*** are *Shared* and *Exclusive*. When set to *Shared*, CoreDX DDS does not enforce unique ownership for each instance, and multiple *DataWriters* can update the same instance (DataReaders will receive the data written by all mached DataWriters). When set to Exclusive, each instance can be modified only by one DataWriter. In this case, one DataWriter "owns" each instance, and while that DataWriter is "alive", matched DataReaders will only accept samples on an instance written by the instance owner.

A DataReader can automatically change the owner of an instance to a different DataWriter. This will happen if the current owner misses a deadline or is otherwise considered to be not actively writing on the instance. The DataReader will then assign ownership to the active DataWriter with the next highest *strength*.

This is a QoS policy that must be compatible before DataReaders and DataWriters will match. The DataReader and DataWriter ***Ownerships*** must match before the DataWriter and DataReader will communicate. If the ***Ownership*** is not compatible, CoreDX DDS will generate an IncompatibleQos status (see the *Communication Status* chapter for additional information).

12.3.12 OWNERSHIP_STRENGTH

The ***Ownership Strength*** QoS policy is applicable only when the ***Ownership*** QoS policy is set to *Exclusive*. Each DataWriter can set its *Strength* with this QoS setting. This strength is used to determine which DataWriter's updates will be received used the subscribing application when more than one DataWriter is writing on that instance.

12.3.13 PARTITION

The ***Partition*** QoS policy allows the application to define logical partitions in a DDS domain. In order for a DataReader to see data published by a DataWriter, their ***Partitions*** must match. A ***Partition*** is a string that may contain a '*' wildcard. Entities may define (and be part of) multiple partitions. The empty string ("") is a valid partition, and will match only another empty string or '*' wildcard partition.

This is a QoS policy that must be compatible before DataReaders and DataWriters will match. If the ***Partitions*** do not match, CoreDX DDS will generate an IncompatibleQos status (see the *Communication Status* chapter for additional information).

12.3.14 PRESENTATION

The ***Presentation*** QoS policy controls the extent to which published data changes are dependent on each other. The possible values for ***Presentation*** are *coherent access* and *ordered access*. In addition, there is an additional configuration item: *Access Scope*.

When ***Presentation*** is configured as *coherent access*, CoreDX DDS will preserve groupings of changes made by the publishing application between calls to the *begin_coherent_change()* and *end_coherent_change()* operations. When *Access Scope* is set to INSTANCE, the scope for grouping changes is within each individual instance. In this case, calls to *begin_coherent_change()* and *end_coherent_change()* have no effect. When *Access Scope* is set to TOPIC, then coherent changes made by a DataWriter will be preserved and made available as a set to each DataReader. When *Access Scope* is set to GROUP, coherent changes made by all DataWriters attached to a Publisher will be preserved and made available to each Subscriber.

When ***Presentation*** is configured as *ordered access*, CoreDX DDS will preserve the order of changes made by the publishing application, in

accordance with the setting of *Access Scope*. When *Access Scope* is set to INSTANCE, changes to an instance are unordered relative to other instances. When *Access Scope* is set to TOPIC, changes made by a single DataWriter are made available to DataReaders in the same order in which they occurred. When *Access Scope* is set to GROUP, changes made by all DataWriters attached to a Publisher are made available to Subscribers in the same order in which they occurred.

Note that while changes are preserved by CoreDX DDS and made available to the DataReaders or Subscribers in order, the application must make the appropriate calls to the DataReader or Subscriber in order to see the data in the desired order.

This is a QoS policy that must be compatible before DataReaders and DataWriters will match.

12.3.15 READER_DATA_LIFECYCLE

The ***Reader Data Lifecycle*** QoS policy controls the behavior of the DataReader with regard to the data it has received and is maintaining. A DataReader internally maintains data samples it has received until they have been 'taken' by the application according to ***History*** and ***Resource Limits*** QoS policy settings. A DataReader will maintain information for an instance, even when the associated DataWriter is no longer alive, until the application has "taken" all samples for that instance.

The ***Reader Data Lifecycle*** QoS policy offers some memory usage protection to the application, by allowing CoreDX DDS to release resources for instances, even if the application neglects to "take" all samples for these instances. This QoS policy offers two configuration items. The *autopurge nowriter samples delay* configuration item defines the amount of time the DataReader will maintain information for an instance once it becomes NOT_ALIVE_NO_WRITERS (there are no live writers writing this instance). The *autopurge disposed samples delay* configuration item defines the amount of time the DataReader will maintain information for an instance once it becomes NOT_ALIVE_DISPOSED (a writer has disposed this instance). CoreDX DDS may reclaim an instance on a DataReader when the instance state is NOT_ALIVE_NO_WRITERS and there are no samples associated with the instance.

12.3.16 RELIABILITY

The ***Reliability*** QoS policy configures the level of reliability CoreDX DDS will guarantee for communications between a DataReader and DataWriter. The possible values for ***Reliability*** are *Best Effort* and *Reliable*.

With a *Best Effort* configuration, CoreDX DDS will make an effort to deliver all published data, but there is no guarantee all data will be received by all DataReaders.

With a *Reliable* configuration, CoreDX DDS will use an additional reliability protocol to check if written samples are received, and possibly resend them if necessary. The *Reliable* configuration may be used in combination with the ***History*** and ***Resource Limits*** QoS policies to guarantee all published data will be received by all matched DataReaders. This configuration requires more resources and overhead to fulfill. The ***History*** and ***Resource Limits*** QoS policies will determine the amount of resources CoreDX DDS will maintain in order to meet *Reliable* requirements. If configured resource limits are met, the publishing application may block on a write() operation.

This is a QoS policy that must be compatible before DataReaders and DataWriters will match. A *Reliable* DataWriter will match any DataReader, *Best Effort* or *Reliable*. A *Best Effort* DataWriter will match only *Best Effort* DataReaders.

12.3.17 RESOURCE_LIMITS

The ***Resource Limits*** QoS policy configures the resources CoreDX DDS can use in order to meet the requirements imposed by the application and other QoS settings (including ***History***, ***Durability***, and ***Reliability***). The ***Resources Limits*** that can be configured include: the total number of samples (*max_samples*), the total number of instances (*max_instances*), and the number of samples in each instance (*max_samples_per_instance*).

CoreDX DDS DataReaders and DataWriters will not store more samples or instance than is specified by their Resource Limits QoS policies. This provides a convenient mechanism to constrain the amount of memory a CoreDX DDS application will use for application data. It also allows CoreDX DDS to pre-allocate memory resources, which can result in better performance.

The Resource Limits QoS policy provides a hard limit for the number of samples or instances that can be stored by a DataReader or DataWriter.

The configuration of other QoS Policies: Reliability, Durability, and History determine the behavior of DataReaders and DataWriters when their Resource Limits are met.

For additional information, refer to the 10.5 Data Cache section, as well as the sections for these specific QoS policies: 12.3.16 *RELIABILITY*, 12.3.3 *DURABILITY*, and 12.3.7 *HISTORY*.

It is possible for a DataWriter to publish data samples (and instances) faster than a DataReader is consuming them, causing the DataReader to fill up its Data Cache to its configured Resource Limits. This could be with respect to *samples* (max_samples or max_samples_per_instance) or *instances* (max_instances). With the right combination of QoS policies (specifically, Reliable Reliability and Keep All History), published samples will be stored by the DataWriter until all DataReaders can accept them. The DataWriter's Cache of data samples will continue to grow until it meets the configured ***Resources Limits***. When this occurs, CoreDX DDS will return an error on the next call to write() (or dispose() or register_instance()). The application may block first, depending on the configuration of the Reliability blocking time. This is a configuration where one 'slow' DataReader can effectively prevent the DataWriter from publishing any new data, affecting all other DataReaders matched to that DataWriter.

The CoreDX DDS constant DDS::LENGTH_UNLIMITED is used to indicate the absence of a particular limit.

The *max_samples* and *max_samples_per_instance* settings must be consistent such that *max_samples* >= *max_samples_per_instance.* In addition, the *max_samples_per_instance* setting must be consistent with the ***History*** *depth*, such that *depth* <= *max_samples_per_instance.* If there is an error in these settings, a create_datareader() operation will fail. A set_qos() operation will return the error DDS::RETCODE_INCONSISTENT_POLICY.

12.3.18 TIME_BASED_FILTER

The ***Time Based Filter*** QoS policy allows the application to indicate a particular DataReader does not necessarily want to see all data samples published for a Topic. In fact, the DataReader would to see, for each instance, at most one data sample every *n* time period. This time period is the *minimum_separation* for the ***Time Based Filter***.

Using the ***Time Based Filter*** QoS policy can reduce the amount of data written by a DataWriter. This is particularly useful in situations where a DataReader cannot keep up with the amount of data published, or where some DataReaders simply do not need all the intermediate data samples published on a Topic.

The ***Time Based Filter*** *minimum_separation* must be consistent with the ***Deadline*** *period*. Setting a *period* < *minimum_separation* is an error. A create_datareader() operation will fail. A set_qos() operation will return the error DDS::RETCODE_INCONSISTENT_POLICY.

12.3.19 TOPIC_DATA

The ***Topic Data*** QoS policy allows the application to attach additional information to created *Entity* objects. This data is not used by CoreDX DDS, and is made available to the application by the ***Built-in Topics***, along with other discovery information. For more information, see the 9.2 *Built-In Topics* section.

12.3.20 TRANSPORT_PRIORITY

CoreDX DDS does not currently support this QoS policy.

12.3.21 USER_DATA

The ***User Data*** QoS policy allows the application to attach additional information to created *Entity* objects. This data is not used by CoreDX DDS, and is made available to the application by the ***Built-in Topics***, along with other discovery information. For more information, see the 9.2 *Built-In Topics* section.

12.3.22 WRITER_DATA_LIFECYCLE

The ***Writer Data Lifecycle*** QoS policy controls the behavior of the DataWriter with regard to the data it has published and is maintaining. This QoS policy allows the application to configure CoreDX DDS to automatically dispose instances when they are unregistered (see the 10.4 *Instance Lifecycles* for additional information).

The ***Writer Data Lifecycle*** QoS policy contains one configuration items: auto-dispose unregistered instances (*autodispose_unregistered_instances*). Setting this configuration item to *TRUE* causes the DataWriter to dispose the

instance each time it is unregistered. Setting this configuration item to *FALSE* will not automatically dispose instances when they are unregistered.

When a DataWriter is deleted, all instances managed by the DataWriter are automatically unregistered. Therefore, only setting the auto-dispose unregistered instances configuration item will ensure the instances managed by a DataWriter are disposed.

Chapter 13 Communication Status

The DDS infrastructure keeps track of a number of statuses and statistics related to data communications. The application may choose to be made aware of some, all, or none of these statuses and statistics.

Each DDS entity has its relevant statuses, as listed in Table 13-1.

Table 13-1: Communication Statuses

Entity	Status Name	Description
Topic	INCONSISTENT_TOPIC	Another, different, Topic exists with the same name as this Topic.
Subscriber	DATA_ON_READERS	Data is available on one or more DataReaders associated with this Subscriber
DataReader	SAMPLE_REJECTED	A received sample has been rejected because of a RESOURCE_LIMITS QoS setting.
	LIVELINESS_CHANGED	One or more DataWriters that were writing data this DataReader was reading has changed its liveliness (becoming active or inactive).
	REQUESTED_DEADLINE_MISSED	A data update for an instance was not received in the expected time interval (configured in the Deadline QoS Policy).
	REQUESTED_INCOMPATIBLE_QOS	A DataWriter was discovered whose Topic matches this DataReader, but whose QoS is incompatible with this DataReader.
	DATA_AVAILABLE	New data is now available to be read.

Entity	Status Name	Description
	SAMPLE_LOST	A sample was lost (never received).
	SUBSCRIPTION_MATCHED	A DataWriter has been discovered that matches the Topic of this DataReader and has a compatible QoS (or a DataWriter that was previously matched is no longer matched).
DataWriter	LIVELINESS_LOST	The liveliness specified in the LIVELINESS QoS was not respected, and DataReaders will consider this DataWriter no longer active.
	OFFERED_DEADLINE_MISSED	A data update was not received in the expected time interval (configured in the Deadline QoS Policy).
	OFFERED_INCOMPATIBLE_QOS	A DataReader was discovered for the same Topic as this DataWriter, but the QoS requested by that DataReader was incompatible with this DataWriter's QoS.
	PUBLICATION_MATCHED	A DataReader has been found that matches the Topic and Qos of this DataWriter (or a DataReader that was previously matched is no longer matched).

Some communication statuses are associated with data being available for the subscribing application. These are referred to as ***read communication statuses***, and include: DATA_ON_READERS and DATA_AVAILABLE. Since these two statuses indicate the reception of data (the real purpose for a Data Distribution Service) they treated a little differently from the other communication statuses, referred to as ***plain communication statuses***.

13.1 Communication Status Details

Each communication status is described in detail below.

13.1.1 Inconsistent Topic Status

The Inconsistent Topic Status is used to inform the application that another Topic has been registered in the Domain (and Partition, if defined) that has the same name as this Topic, but a different data type. The CoreDX DDS middleware will allow an application to create multiple Topics of the same name and different types. The Inconsistent Topic Status allows applications to be made aware of these inconsistencies.

```
Type:                 Plain Communication Status
Associated Entity:    Topic
Mask Name:            INCONSISTENT_TOPIC_STATUS
Struct Type Name:     InconsistentTopicStatus
```

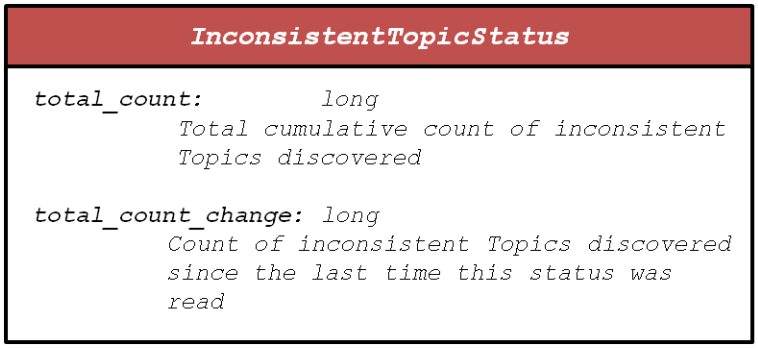

Figure 13-1: Inconsistent Topic Status Structure

13.1.2 Data On Readers Status

The Data On Readers Status is one of the read communication statuses, and is used to inform the application that one or more of the DataReaders attached to a Subscriber has new data samples or sample information.

The Data Available Status and Data On Readers Status are communicated to the application together. In other words, if there is Data Available for a DataReader, then there is a DataReader with new data.

```
Type:                 Read Communication Status
Associated Entity:    Subscriber
Mask Name:            DATA_ON_READERS_STATUS
```

```
Struct Type Name:     N/A
```

13.1.3 Sample Rejected Status

The Sample Rejected Status is used to inform the application that a data sample was not accepted by the DataReader because of resources limits set in an associated QoS (either on the Topic or DataReader). The RESOURCE_LIMITS QoS allows the application to set a limit on the total number of samples, the total number of instances, and the number of samples per each instance kept by the CoreDX DDS middleware. If a DataReader receives a sample that would put any of these numbers over their set limit, the sample is rejected (not added to the reader cache), and the appropriate count on the Sample Reject Status is updated.

```
Type:                 Plain Communication Status
Associated Entity:    DataReader
Mask Name:            SAMPLE_REJECTED_STATUS
Struct Type Name:     SampleRejectedStatus
```

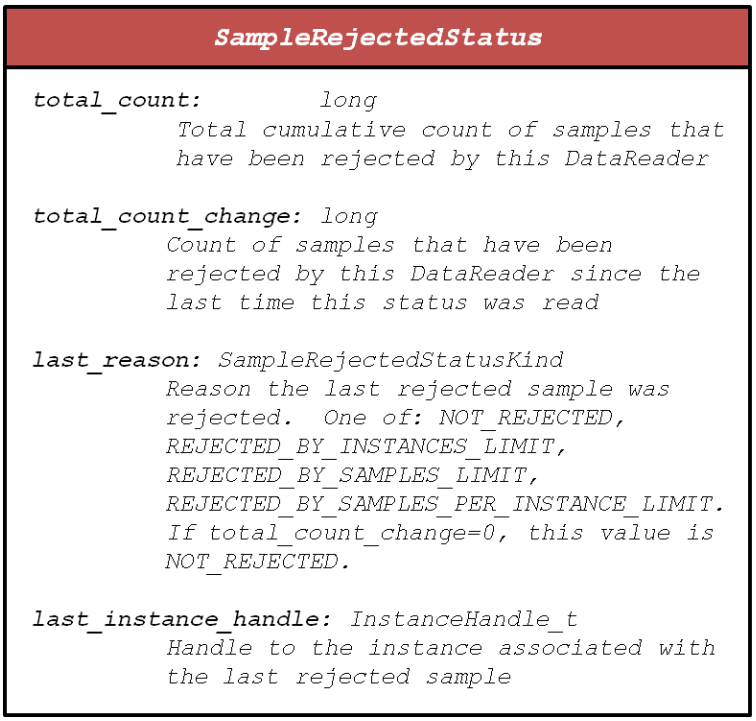

Figure 13-2: Sample Rejected Status Structure

13.1.4 Liveliness Changed Status

The Liveliness Changed Status is used to inform the application of changes to DataWriters known by this DataReader. DataReaders keep track of all DataWriters they are 'matched' with. These matched DataWriters may be ACTIVE (they are either actively writing data or otherwise asserting their liveliness) or INACTIVE (they are not actively writing or asserting their liveliness). Any time one of the DataWriters this DataReader is matched with changes between these states, the Liveliness Changed Status is updated.

```
Type:                   Plain Communication Status
Associated Entity:      DataReader
Mask Name:              LIVELINESS_CHANGED_STATUS
Struct Type Name:       LivelinessChangedStatus
```

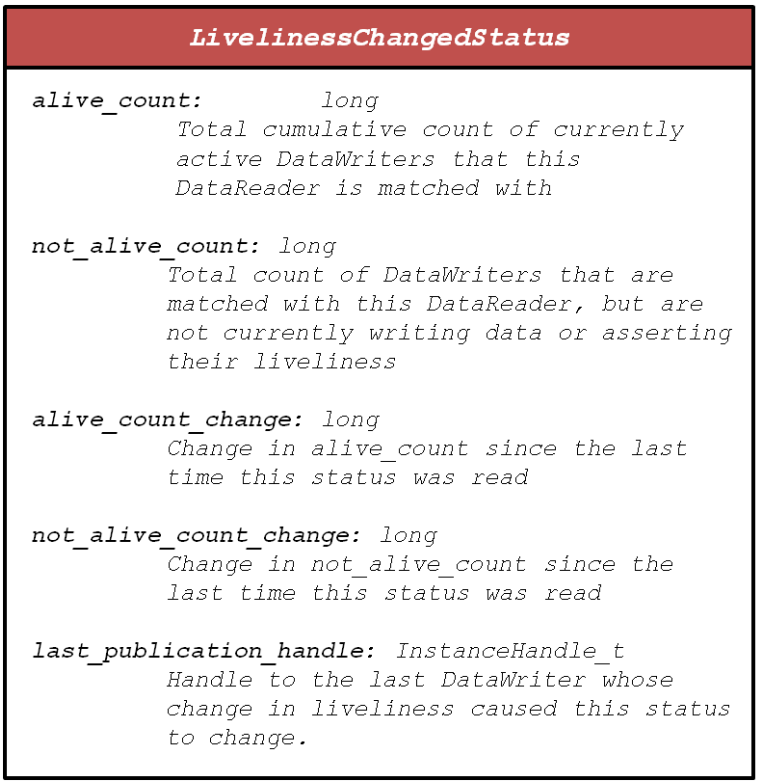

Figure 13-3: Liveliness Changed Status Structure

13.1.5 Requested Deadline Missed Status

The Requested Deadline Missed Status is used to inform the application when a deadline period specified in an associated Deadline QoS (either Topic or DataReader) was missed. The REQUESTED_DEADLINE Qos allows the application to request that an instance be updated at least once every time interval specified by the QoS. When a DataWriter fails to update an instance within the time interval specified by the DataReader's Deadline QoS policy, the Requested Deadline Missed Status is updated.

```
Type:                   Plain Communication Status
Associated Entity:      DataReader
Mask Name:              REQUESTED_DEADLINE_MISSED_STATUS
Struct Type Name:       RequestedDeadlineMissedStatus
```

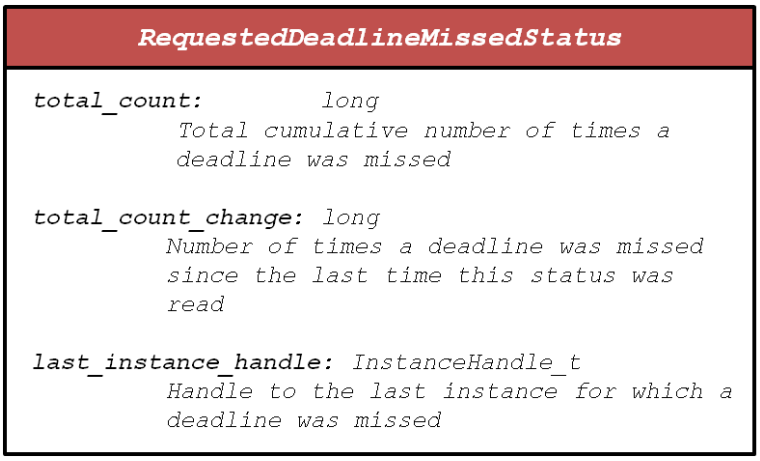

Figure 13-4: Requested Deadline Missed Status Structure

13.1.6 Requested Incompatible QoS Status

The Requested Incompatible QoS Status is used to inform the application that a DataWriter was discovered with a matching Topic and matching data type, but with QoS incompatible to this DataReader's requested QoS. For additional information about compatible and incompatible QoS, see the *Quality of Service Features* chapter.

```
Type:                   Plain Communication Status
Associated Entity:      DataReader
Mask Name:              REQUESTED_INCOMPATIBLE_QOS_STATUS
```

```
Struct Type Name:     RequestedIncompatibleQosStatus
```

RequestedIncompatibleQosStatus

total_count: *long*
Total cumulative number of times a DataWriter has been discovered with a QoS incompatible with this DataReader

total_count_change: *long*
Number of times a DataWriter has been discovered with a QoS incompatible with this DataReader since the last time this status was read

last_policy_id: *QosPolicyId_t*
Id of one of the policies that was inconsistant the last time a DataWriter was discovered with an incompatible QoS

policies[]: *QosPolicyCount*
List of counts for each QoS Policy Id of the number of times an incompatible DataWriter has been found where this Policy Id was incompatible with this DataReader

Figure 13-5: Requested Incompatible QoS Status Structure

13.1.7 Data Available Status

The Data Available Status is one of the read communication statuses (along with the Data On Readers Status), and is used to inform the application of new data available to be read on the associated DataReader.

These two read communication statuses are communicated to the application together. In other words, if there is Data Available for a DataReader, then there is a DataReader with new data.

```
Type:                 Read Communication Status
Associated Entity:    DataReader
Mask Name:            DATA_AVAILABLE_STATUS
Struct Type Name:     N/A
```

13.1.8 Sample Lost Status

The Sample Lost Status is used to inform the application that one or more data samples were not received by a Reader. A sample can be 'lost' for many reasons.

For example, the transport might drop the sample due to congestion or some other reason. If the Reader is not using BEST_EFFORT reliability, then samples dropped by the underlying transport will not be retransmitted, and they are LOST.

Even if reliability is set to RELIABLE, it is still possible to experience lost samples due to other QoS settings. For example, if the Writer is configured to keep very little historical data in its cache (either through HISTORY or RESOURCE_LIMITS QoS), then it is possible that a Reader will fail to get a sample simply because the sample is purged from the Writer's cache before it could be successfully transmitted to the Reader.

```
Type:                   Plain Communication Status
Associated Entity:      DataReader
Mask Name:              SAMPLE_LOST_STATUS
Struct Type Name:       SampleLostStatus
```

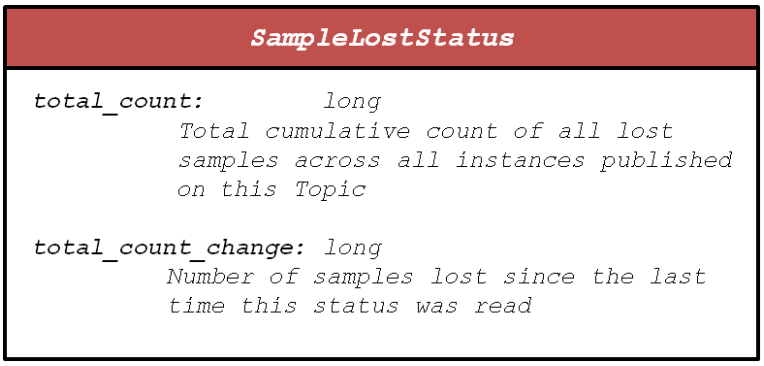

Figure 13-6: Sample Lost Status Structure

13.1.9 Subscription Matched Status

The Subscription Matched Status is used to inform the application that a new DataWriter has been discovered that matches this DataReader's QoS settings and it is producing data this DataReader is interested in. That is,

the DataWriter is writing data on the same Topic this DataReader is reading on and its QoS settings are compatible with this DataReader.

```
Type:                   Plain Communication Status
Associated Entity:      DataReader
Mask Name:              SUBSCRIPTION_MATCHED_STATUS
Struct Type Name:       SubscriptionMatchedStatus
```

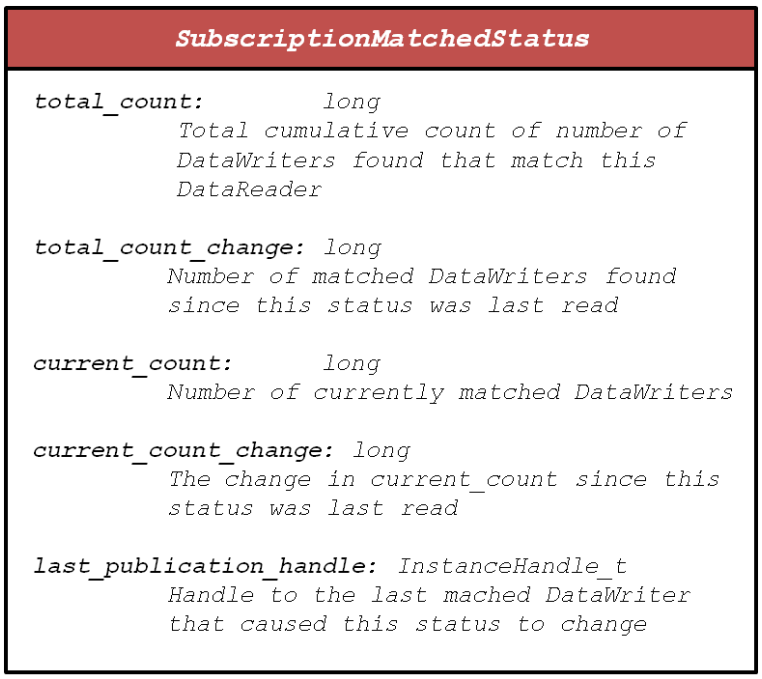

Figure 13-7: Subscription Matched Status Structure

13.1.10 Liveliness Lost Status

The Liveliness Lost Status is used to inform the application that this DataWriter has missed asserting its liveliness in the time period specified by its LIVELINESS QoS policy. The DataWriter's LIVELINESS QoS policy allows the publishing application to specify the interval in which this DataWriter will either write a sample or assert its liveliness to all matched DataReaders. If a DataWriter misses this specified window, the Liveliness Lost Status is updated.

```
Type:                   Plain Communication Status
```

```
Associated Entity:    DataWriter
Mask Name:            LIVELINESS_LOST_STATUS
Struct Type Name:     LivelinessLostStatus
```

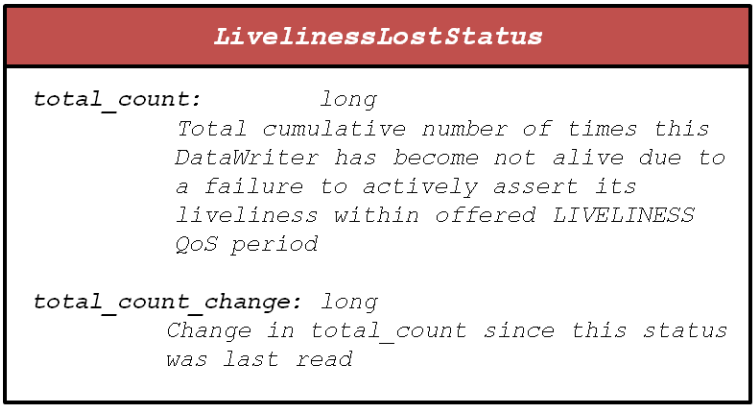

Figure 13-8: Liveliness Lost Status Structure

13.1.11 Offered Deadline Missed Status

The Offered Deadline Missed Status is used to inform the application when a deadline period specified in an associated Deadline QoS (either Topic or DataWriter) was missed. The OFFERED_DEADLINE Qos allows the application to commit to updating each instance at least once every time interval specified by the QoS setting. When the DataWriter fails to update an instance within the specified time interval, the Offered Deadline Missed Status is updated.

```
Type:                 Plain Communication Status
Associated Entity:    DataWriter
Mask Name:            OFFERED_DEADLINE_MISSED_STATUS
Struct Type Name:     OfferedDeadlineMissedStatus
```

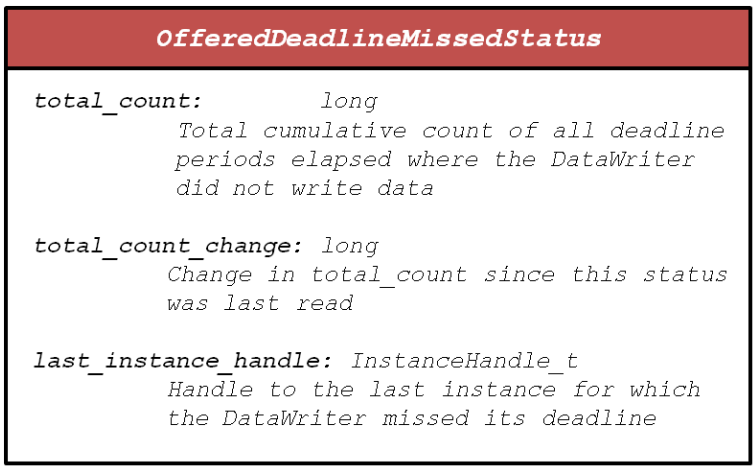

Figure 13-9: Offered Deadline Missed Status Structure

13.1.12 Offered Incompatible QoS Status

The Offered Incompatible QoS Status is used to inform the application that a DataReader was discovered with a matching Topic and matching data type, but with QoS incompatible to this DataWriter's offered QoS. For additional information about compatible and incompatible QoS, see the *Quality of Service Features* chapter.

Type:	Plain Communication Status
Associated Entity:	DataWriter
Mask Name:	OFFERED_INCOMPATIBLE_QOS_STATUS
Struct Type Name:	OfferedIncompatibleQosStatus

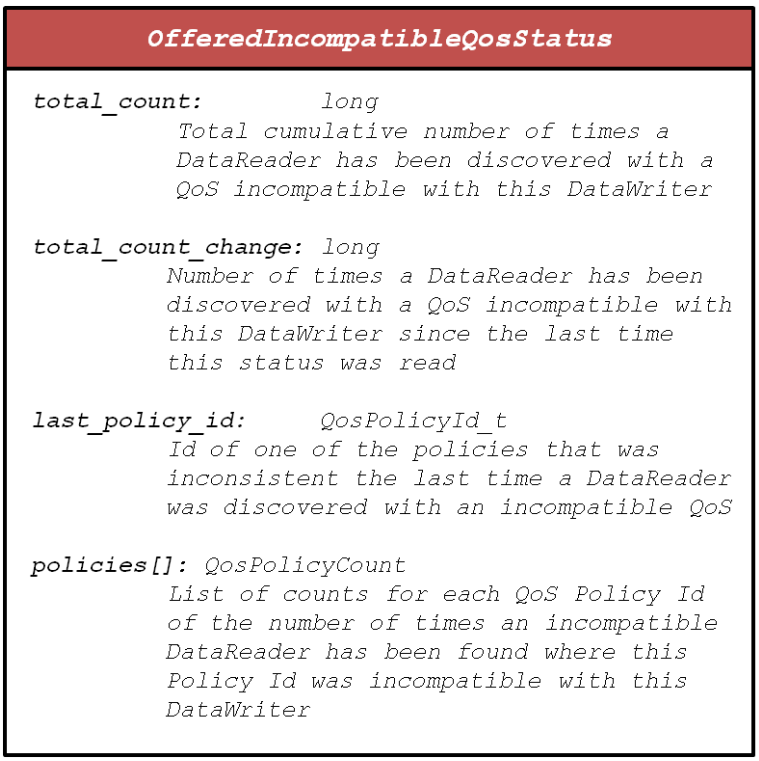

Figure 13-10: Offered Incompatible QoS Status Structure

13.1.13 Publication Matched Status

The Publication Matched Status is used to inform the application that a new DataReader has been discovered that matches this DataWriter's QoS settings.

```
Type:                 Plain Communication Status
Associated Entity:    DataWriter
Mask Name:            PUBLICATION_MATCHED_STATUS
Struct Type Name:     PublicationMatchedStatus
```

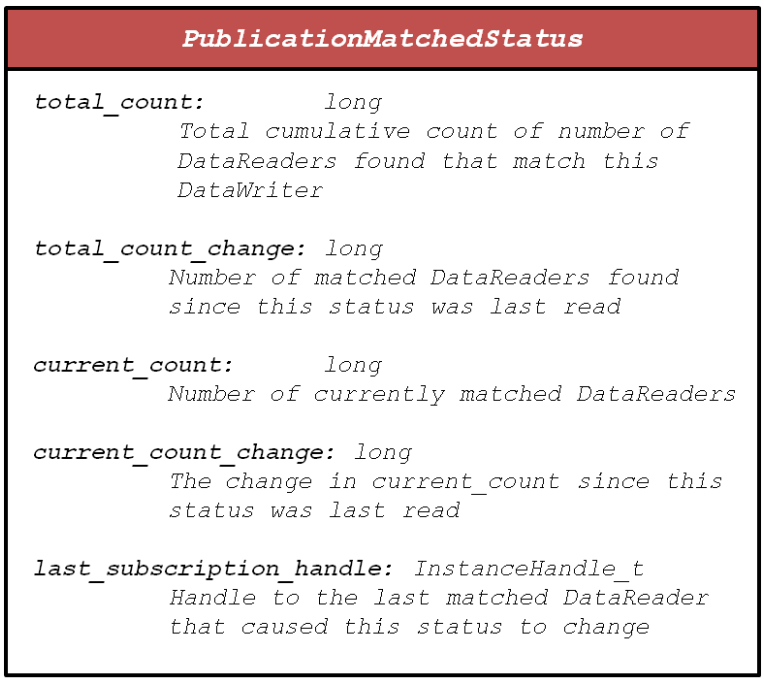

Figure 13-11: Publication Matched Status Structure

13.2 Application Access to Communication Status

An application can access ***plain communication statuses*** associated with an entity by calling that entity's get_<status_name>() method. For example, a DataReader will have a get_sample_lost_status() method to obtain a snapshot of its SampleLostStatus, and a Topic will have a get_inconsistent_topic_status() method to obtain a snapshot of its InconsistentTopicStatus.

Read communication status are used to indicate the availability of data. The data may then accessed by calling the DataReader read() or take() methods. The application may call read() or take() directly at any time, even if there is no data available.

While the application can choose to call get_xxx_status(), read(), or take() at any time, typically the application will wait for notification from the infrastructure that a status has changed (or data is available) before accessing the status information or data.

There are two mechanisms an application may use to learn about changes in communication statuses and statistics. The first is ***listeners***, where an application can asynchronously determine and handle a change in communication statuses. The second is ***conditions*** (using wait sets), where an application can block waiting for a status change.

13.2.1 Listeners

Listeners provide an asynchronous method for the application to be notified of changes in statuses. The application provides a hook for the CoreDX DDS middleware to invoke upon a particular status change. For example, an application interested in the Data Available Status for its DataReader will provide an on_data_available() method to the CoreDX DDS middleware, and the CoreDX DDS middleware will call the provided method when new data is available on that DataReader.

All DDS entities support a listener, and all listeners have a type specific to their associated entity. For example, the DataReaderListener is associated with a DataReader.

Listeners are interfaces. Each listener provides a set of methods that correspond to the relevant communication statuses for that entity.

Listeners are hierarchical. *Figure 13-12: Listener Hierarchy* depicts the hierarchy of all the listeners.

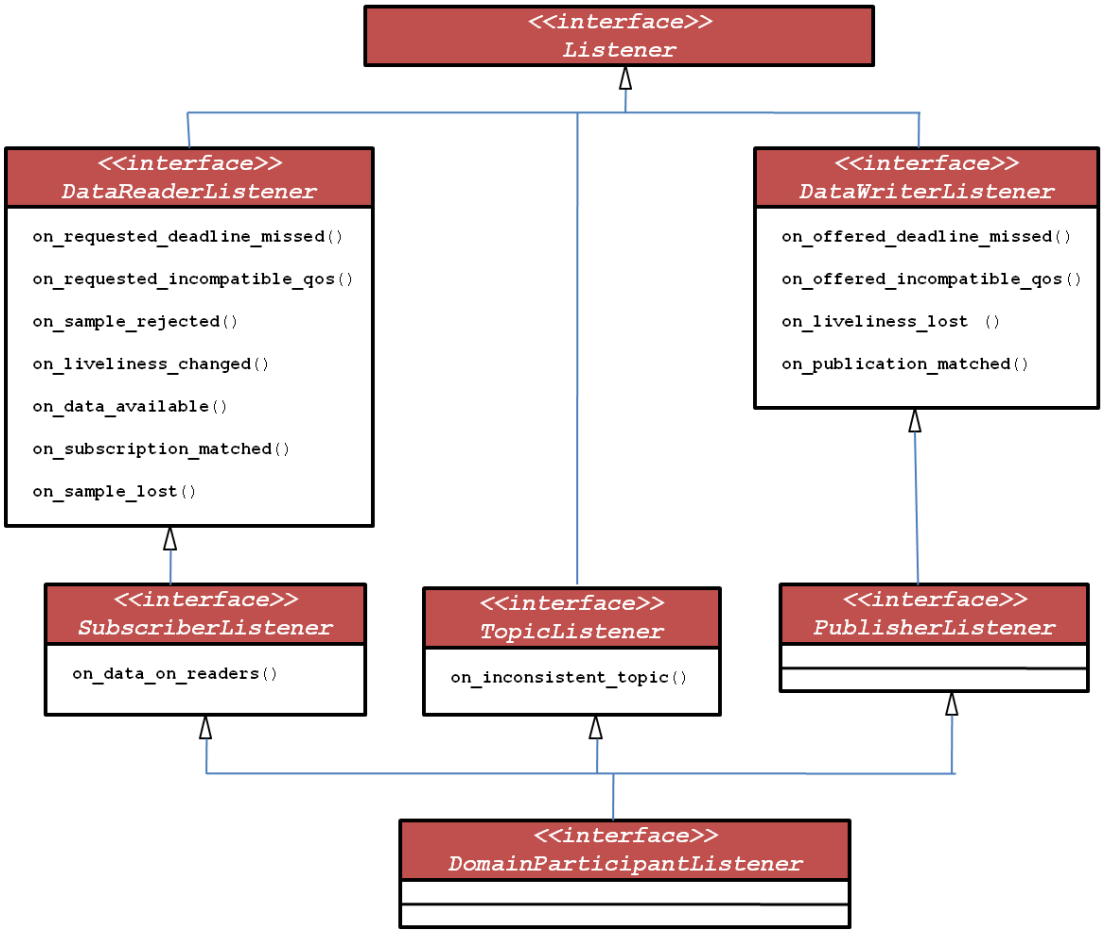

Figure 13-12: Listener Hierarchy

The application must implement an appropriate listener interface in order to receive communication status changes. The following Table 13-2 depicts the listener methods for each entity.

Table 13-2: Listener Method Signatures

Entity	Listener Method Signature
DataReaderListener	void **on_requested_deadline_missed**(DataReader, RequestedDeadlineMissedStatus); void **on_requested_incompatible_qos**(DataReader, RequestedIncompatibleQosStatus); void **on_sample_rejected**(DataReader, SampleRejectedStatus); void **on_liveliness_changed**(DataReader, SampleRejectedStatus); void **on_data_available**(DataReader); void **on_subscription_matched**(DataReader, SubscriptionMatchedStatus); void **on_sample_lost**(DataReader, SampleLostStatus);
SubscriberListener	void **on_data_on_readers**(Subscriber); (inherits all DataReaderListener methods)
TopicListener	void **on_inconsistent_topic**(Topic, InconsistentTopicStatus);
DataWriterListener	void **on_liveliness_lost**(DataWriter, LivelinessLostStatus); void **on_offered_deadline_missed**(DataWriter, OfferedDeadlineMissedStatus); void **on_offered_incompatible_qos**(DataWriter, OfferedIncompatibleQosStatus); void **on_publication_matched**(DataWriter, PublicationMatchedStatus);
PublisherListener	(inherits all DataWriterListener methods)
DomainParticipantListener	(inherits all SubscriberListener, PublisherListener, and TopicListener methods)

Notice that the listener methods for plain communication statuses follow the same format: they return a void and take the entity and appropriate status as arguments. The listener methods for read communication statuses are a little different. Read communication statuses do not have associated status structures. The only argument is the concerned DataReader (for the on_data_available() method) and Subscriber (for the on_data_on_readres() method). It is assumed that in handling the read communication status, the application intends to eventually read the available data, and providing the appropriate DataReader or Subscriber allows the application to do this.

When the application attaches a listener to an entity, it must also set a mask that indicates which listener methods are enabled within this listener.

13.2.1.1 Listener Access to Plain Communication Statuses

Notice in *Figure 13-12* that the listeners form a hierarchy. When a plain communication status changes, the middleware will invoke the most specific relevant listener method that is enabled.

For example, consider the PublicationMatchedStatus. The corresponding on_publication_matched() listener method comes from the DataWriterListener, and is inherited by the PublisherListener and the DomainParticipantListener. When there is a change to the PublicationMatchedStatus, the DDS infrastructure will look for an enabled on_publication_matched() listener method to invoke. It will look at the DataWriterListener first. If there is not an enabled on_publication_matched() listener method, it will then look at the PublisherListener. If there is not an enabled on_publication_matched() listener method, it will look at the DomainParticipantListener. The first enabled listener method will be invoked. If there are no listeners enabled, no listener methods will be invoked. The status is still available and may trigger a configured Condition, or be accessed by calling the associated get_xxx_status().

13.2.1.2 Listener Access to Read Communication Statuses

The read communication status listeners are invoked differently than plain communication status listeners. The two read communication statuses constitute the real purpose of the Data Distribution Service, and require special consideration.

Each time a read communication status changes the DDS perform the following actions.

1. The infrastructure will attempt to invoke the on_data_on_readers() method on the SusbscriberListener with a parameter of the related Subscriber
2. If this doesn't work (either there was no listener installed or the method was not enabled via the listener mask), the DDS middleware will attempt to trigger the on_data_available() method on the related DataReaderListener with a parameter of the related DataReader.

13.2.1.3 Nil Listeners

The application can choose to install a ***nil listener*** in place of any listener method. When the infrastructure finds a nil listener, it will perform a NO-OP operation and stop looking for enabled listener methods.

13.2.1.4 Implementing Listeners in C

Listeners in C are implemented as a structure of function pointers. In order to implement a listener method, the application must write a function, and then assign it to the appropriate listener function pointer.

For example, a DataReader interested in only the DATA_AVAILABLE status might do the following:

on_data_available Listener

```
void my_on_data_available( DDS_DataReader dr )
{
     Printf("Data is available!\n");
     /* process data by calling DDS_DataReader_read()
      * or DDS_DataReader_take()
      */
}

DDS_DataReaderListener drListener =
{
    NULL, NULL, NULL, NULL,
    my_on_data_available,
    NULL, NULL
}

/* when we create the DataReader */
Dr = DDS_Subscriber_create_datareader( sub,
            DDS_Topic_TopicDescription(topic), &drListener,
            DDS_DATA_AVAILABLE_STATUS);
```

Figure 13-13: Listener Example C Code

The last argument to the create_datareader() method is the listener status mask, telling the DDS middleware which listener methods are enabled.

In C, a nil listener is installed simply by setting the appropriate function pointer to NULL, and then setting that status in the listener mask. Suppose in the above example, the listener mask used in the create_datareader() call was:

```
DDS_DATA_AVAILABLE_STATUS |
DDS_LIVELINESS_CHANGED_STATUS
```

Then there would be a nil listener installed for the on_liveliness_changed() method, and an actual listener installed for the on_data_available() method. All other listener methods would default "down" the listener hierarchy, looking for enabled corresponding listener methods on the Subscriber, and then the DomainParticipant.

The definition of all the C listeners can be found in the CoreDX DDS header file (DDS_HOME/include/dds/core/dds.h or dds.hh).

13.2.1.5 Implementing Listeners in C++

Listeners in C++ are classes containing virtual on_<communication_status>() methods. In order to implement a listener method, the application must create a listener class that derives from the appropriate virtual listener class, and then implement the desired listener method.

For example a DataReader who is only interested in the DATA_AVAILABLE status might do the following:

on_data_available Listener

```
class MyDRListener : public DataReaderListener
{
   public:
      void on_data_available( DataReader * dr );
};

void MyDRListener::on_data_available(DataReader * dr)
{
    printf("Data Available!\n");
```

```
        /* process data by calling read() or take()
         */
    }

    /* when we create the DataReader */
    MyDRListener drListener;
    Dr = sub->create_datareader( (TopicDescription*)topic,
    DATAREADER_QOS_DEFAULT, &drListener, DATA_AVAILABLE_STATUS );
```

Figure 13-14: Listener Exampe C++ Code

To install a nil listener in C++, use the nil_listeners member on the appropriate listener object. For example, to add a nil listener for the on_liveliness_changed() listener method:

```
drListener . nil_listeners = LIVELINESS_CHANGED_STATUS;
```

This indicates to the DDS middleware that the on_liveliness_changed() listener method should be treated as a nil listener method.

13.2.1.6 Implementing Listeners in C#

Listeners in C# are classes that may be used as a base class for application data specific Listeners. In order to implement a listener method, the application must create a listener class that derives from the appropriate listener base class, and then implement the desired listener method.

For example a DataReader who is only interested in the DATA_AVAILABLE status might do the following:

on_data_available Listener

```
class MyDRListener : DataReaderListener
{
   Public MyDRListener()
   {
      this.on_requested_deadline_missed  = null;
      this.on_requested_incompatible_qos = null;
      this.on_sample_rejected            = null;
      this.on_liveliness_changed         = null;
      this.on_data_available             = data_available;
      this.on_subscription_matched       = null;
      this.on_sample_lost                = null;
   }

public void data_available( DataReader dr )
{
```

```
        System.Console.WriteLine ("Data Available!\n");
        /* process data by calling read() or take()
         */
    }

    /* when we create the DataReader */
    MyDRListener drListener = new MyDRListener();
    Dr = sub->create_datareader( topic, DDS.DATAREADER_QOS_DEFAULT,
    drListener, DDS.DATA_AVAILABLE_STATUS );
```

The last argument to the create_datareader() method is the listener status mask, telling the DDS middleware which listener methods are enabled.

In C#, a nil listener is installed simply by setting the appropriate function pointer to NULL, and then setting that status in the listener mask. Suppose in the above example, the listener mask used in the create_datareader() call was:

```
DDS.DATA_AVAILABLE_STATUS |
DDS.LIVELINESS_CHANGED_STATUS
```

Then there would be a nil listener installed for the on_liveliness_changed() method, and an actual listener installed for the on_data_available() method. All other listener methods would default "down" the listener hierarchy, looking for enabled corresponding listener methods on the Subscriber, and then the DomainParticipant.

13.2.1.7 Implementing Listeners in Java

Listeners in Java are interfaces containing empty on_<communication_status>() methods. In order to implement a listener method, the application must create a listener class that implements the appropriate listener interface, and then create public versions of all listener methods. Listener methods may be empty.

For example a DataReader who is only interested in the DATA_AVAILABLE status might do the following:

on_data_available Listener

```
    class MyDRListener implements DataReaderListener
    {
        public long get_nil_mask() { return 0; }
```

```
        public void on_data_available(DataReader dr)
        {
          System.out.println(" DATA AVAILABLE ");
      /* process data by calling read() or take()  */
        }

        public void on_requested_deadline_missed(DataReader dr,
                    RequestedDeadlineMissedStatus status) { };
        public void on_requested_incompatible_qos(DataReader dr,
                    RequestedIncompatibleQosStatus status) { };
        public void on_sample_rejected (DataReader dr,
                    SampleRejectedStatus status) { };
        public void on_liveliness_changed (DataReader dr,
                    LivelinessChangedStatus status) { };
        public void on_subscription_matched (DataReader dr,
                    SubscriptionMatchedStatusstatus) { };
        public void on_sample_lost(DataReader dr,
                    SampleLostStatus status) { };

/* when we create the DataReader */
DataReaderListener dr_listener = new MyDRListener();
dr = sub.create_datareader( topic, DDS.DATAREADER_QOS_DEFAULT,
drListener, DDS.DATA_AVAILABLE_STATUS );
```

To install a nil listener in Java, implement the get_nil_maks() method on the appropriate listener object to return the corresponding status. For example, to add a nil listener for the on_liveliness_changed() listener method:

```
public long get_nil_mask() { return
                    DDS.LIVELINESS_CHANGED_STATUS; }
```

This indicates to the DDS middleware that the on_liveliness_changed() listener method should be treated as a nil listener method.

13.2.2 Conditions and WaitSets

The listener notification method is asynchronous. Conditions and WaitSets provide a wait-based mechanism to be notified of changes in the CoreDX DDS infrastructure. In general, the application using Conditions and WaitSets will use the following pattern.

1. The application indicates which statuses it is interested in by obtaining or creating ***Condition*** objects and attaching them to a ***WaitSet***.
2. The application waits on the *WaitSet* until the trigger value of one or more of the attached *Condition* objects becomes TRUE.
3. The application calls get_status_changes() to determine the what changed.
4. The application calls the appropriate get_<communication_status>(), read(), or take() method(s).

Conditions are always used in combination with a WaitSet. The Condition contains a trigger value that is set when there is a change (change in communication status, change in read status, for example). The WaitSet blocks the application until the Condition's trigger value is set (or until an application-defined timeout is reached). There are different kinds of Conditions available for the application to use. These are described in the sections below.

13.2.2.1 StatusConditions

StatusConditions allow the application to access plain communication statuses. A Condition can be obtained from each entity that contains a communication status, by using the get_status_condition() operation.

A StatusCondition contains a mask of enabled statuses. Similar to the listener mask, this mask allows the application to tailor the Condition to trigger only for specific status changes. For example, a DataWriter contains four statuses: liveliness_lost, offered_deadline_missed, offered_incompatible_qos, and publication_matched. The StatusCondition returned from DataWriter::get_statuscondition() will have all four statuses enabled by default. If any one of these statuses changes for the DataWriter, the Condition will be set, and the application waiting on the corresponding WaitSet will be signaled. The application can use the enabled statuses mask to enable only the communications statuses that are of interest to the application.

13.2.2.2 ReadConditions and QueryConditions

ReadConditions and QueryConditions allow the application to be notified when data is available. Specifically, these Conditions allow the application to be notified when a specific *kind* of data is available. The ReadCondition allows the application to configure the view_states, instance_states, and sample_states that the ReadCondition should trigger on. The QueryCondition allows the application to define the content of the data

samples that should trigger the QueryCondition. This is done using an sql-like query string. GuardConditions

GuardConditions allow the application to control triggering the condition. GuardConditions are not attached to a CoreDX DDS entity (DataReader, Topic, etc), rather they can be used by the application for synchronization efforts outside the DDS middleware. Because these conditions are not attached to a CoreDX DDS entity, they are created by using the GuardCondition__alloc() operation (C interface), or the GuardCondition constructor (C++, C#, Java interfaces). The application is responsible for releasing the GuardCondition memory.

13.2.2.3 Implementing Conditions in C

Condition Examples

```
/* Variabled used in sample code */
DDS_DataReader  dr;
DDS_StatusMask sm;
DDS_StatusCondition sc;
DDS_ReadCondition rc;
DDS_WaitSet ws;
DDS_ConditionSeq active_conditions;
DDS_Duration_t timed;
DDS_ReturnCode_t retval;

/* Create DomainParticipant, register data type,
 * create Topic, Subscriber - code not shown here.
 */

/* Create DataReader with no Listeners */
dr = DDS_Subscriber_create_datareader( sub,
                            DDS_TopicDescription(topic),
                            DDS_DATAREADER_QOS_DEFAULT,
                            NULL, 0);

/* Get the StatusCondition, configure to only trigger on
 * subscription matched status changes
 */
sc     = DDS_DataReader_get_statuscondition( dr );
sm     = DDS_SUBSCRIPTION_MATCHED_STATUS;
retval = DDS_StatusCondition_set_enabled_statuses( sc, sm );
```

```
/* Create the ReadCondition, configure to trigger
 * only on samples not yet read
 */
rc = DDS_DataReader_create_readcondition( dr,
                                    DDS_NOT_READ_SAMPLE_STATE,
                                    DDS_ANY_VIEW_STATE,
                                    DDS_ANY_INSTANCE_STATE );

/* Create a WaitSet and attach all my conditions */
ws = DDS_WaitSet__init( DDS_WaitSet__alloc() );
retval = DDS_WaitSet_attach_condition( ws, sc );
retval = DDS_WaitSet_attach_condition( ws, rc );

/* Wait for something to happen (no timeout) */
INIT_SEQ(active_conditions);
timed.sec = 0;      /* infinite */
timed.nanosec = 0;
retval = DDS_WaitSet_wait( ws, &active_conditions, &timed );

/* Something woke us up -- check */
for (i=0 ; i<seq_get_length(&active_conditions) ; i++)
{
   if ( active_conditions._buffer[i] == (DDS_Condition)sc )
   {
       /* If you don't already know what entity this
        * StatusCondition is attached to, here's how
        * to find out.
        */
       DDS_Entity e = DDS_StatusCondition_get_entity(sc);
       DDS_DataReader r = (DDS_DataReader)e;

       DDS_StatusMask s = DDS_DataReader_get_status_changes(r);
       if ( s & DDS_SUBSCRIPTION_MATCHED_STATUS )
           /* Handle subscription match */
   }

   else if ( active_conditions._buffer[i] == (DDS_Condition)rc )
   {
       /* handle data available on our DataReader 'dr' */
   }
}

/* Cleanup */
retval = DDS_DataReader_delete_readcondition( dr, rc );
```

Figure 13-15: Condition Example C code

13.2.2.4 Implementing Conditions in C++

Condition Examples

```
using namespace DDS;

/* Variabled used in sample code */
DataReader *      dr;
StatusMask        sm;
StatusCondition * sc;
ReadCondition *   rc;
WaitSet           ws;
ConditionSeq      active_conditions;
Duration_t       timed;
ReturnCode_t     retval;

/* Create DomainParticipant, register data type,
 * create Topic, Subscriber – code not shown here.
 */

/* Create DataReader with no Listeners */
dr = sub->create_datareader( (TopicDescription)topic,
                             DATAREADER_QOS_DEFAULT,
                             NULL, 0);

/* Get the StatusCondition, configure to only trigger on
 * subscription matched status changes
 */
sc     = dr -> get_statuscondition();
sm     = SUBSCRIPTION_MATCHED_STATUS;
retval = sc -> set_enabled_statuses( sm );

/* Create the ReadCondition, configure to trigger
 * only on samples not yet read
 */
rc = dr -> create_readcondition(  NOT_READ_SAMPLE_STATE,
                                  ANY_VIEW_STATE,
                                  ANY_INSTANCE_STATE );

/* Create a WaitSet and attach all my conditions */
retval = ws . attach_condition( sc );
retval = ws . attach_condition( rc );

/* Wait for something to happen (no timeout) */
timed.sec = 0;      /* infinite */
timed.nanosec = 0;
retval = ws . wait( &active_conditions, &timed );

/* Something woke us up -- check */
for (i=0 ; i<active_conditions . size() ; i++)
{
```

```
        if ( active_conditions[i] == (Condition)sc )
        {
            /* If you don't already know what entity this
             * StatusCondition is attached to, here's how
             * to find out.
             */
            Entity e     = sc -> get_entity();
            DataReader r = (DataReader)e;

            StatusMask s = r -> get_status_changes();
            if ( s & SUBSCRIPTION_MATCHED_STATUS )
                /* Handle subscription match */
        }

        else if ( active_conditions[i] == (Condition)rc )
        {
            /* handle data available on our DataReader 'dr' */
        }
    }

    /* Cleanup */
    retval = dr -> delete_readcondition( dr, rc );
```

Figure 13-16: Condition Example C++ code

13.2.3 Using Listeners and Conditions in Combination

The application may choose to use both listeners and conditions in combination. One way to do this is using listeners for some communication statuses and using conditions for other communication status. However, if both listeners and conditions are used for an individual communication status, the listener method is invoked first and then the condition objects are signaled. Since calling a listener has the effect of "resetting" the status, when the Condition is signaled the corresponding status will not be set.

Part 4: CoreDX DDS Extensions

This section introduces some CoreDX DDS specific extensions to the DDS API. These include facilities for logging, transports, licensing, adjusting the discovery mechanisms, naming DDS entities, and other concepts that make using DDS easier.

Chapter 14 CoreDX DDS Logging

This chapter describes the logging facilities provided by CoreDX DDS and how to configure them.

First, it should be noted that there are two versions of CoreDX DDS libraries provided. The first is the optimized, performance focused library which does not contain the extra logging instrumentation code. This library is faster and smaller than the instrumented library. This library contains very little logging, primarily limited to indicating error or anomalous conditions.

The second version of libraries includes a rich set of logging instrumentation. During application development and debugging, it may be useful to link against the logging version of the CoreDX DDS libraries. Then, deployed applications can be linked with the streamlined libraries for enhanced performance and resource utilization.

For C# and Java users: The coredx_csharp.dll (C#) and coredx_dds.jar (java) packages refer to the non-logging native library name (dds_csharp.dll/lib for C# and dds_java.so/dll/lib for java). In order to use the logging versions of these libraries, these libraries must be renamed (or linked to). For example, to use the Java logging library under LInux:

```
% cd ${COREDX_TOP}/target/${TARGET_ARCH}
% ls –l libdds_java*
      -rw-r--r-- 1 user grp libdds_java_log.so
      -rw-r--r-- 1 user grp libdds_java.so
% mv libdds_java.so libdds_nolog.so
% ln –s libdds_java_log.so libdds_java.so
% ls –l libdds_java*
      -rw-r--r-- 1 user grp libdds_java_log.so
      -rw-r--r-- 1 user grp libdds_java_nolog.so
      -rw-r--r-- 1 user grp libdds_java.so ->
             libdds_java_log.so
```

Once the application has been linked with the instrumented library, the logging output can be controlled either by environment variable or by adjusting the CoreDX_LoggingQosPolicy.

Controlling logging with an environment variable is quick and easy, but does not offer fine-grained control. The Logging QoS policy offers greater control.

Whichever mechanism is used, the level of logging output is controlled by a series of bitmapped flags. Each class of log message is enabled by setting a flag to 1, and is disabled by 0.

Table 14-1: CoreDX DDS Logging Flags

CoreDX DDS Debug Categories and Flags	
COREDX_ERROR_LOGGING_QOS	0x0001
COREDX_DATA_LOGGING_QOS	0x0002
COREDX_DISCOVERY_LOGGING_QOS	0x0004
COREDX_FACTORY_LOGGING_QOS	0x0008
COREDX_LIVELINESS_LOGGING_QOS	0x0010
COREDX_STATUS_LOGGING_QOS	0x0020
COREDX_TRANSPORT_LOGGING_QOS	0x0040

Currently, all logging output is directed to **stderr**. There is an additional field in the CoreDX_LoggingQosPolicy (**uri**) that will be used to direct logging output to other locations, sockets, dds topics, etc in a future CoreDX DDS release. Currently, this field is unused.

Table 14-2: Logging QoS Configuration Example

Setting CoreDX DDS Logging Flags

```
DDS_Subscriber           sub;
DDS_DataReader           dr;
DDS_DataReaderQos        dr_qos;

DDS_Subscriber_get_default_datareader_qos(sub, &dr_qos);
dr_qos.logging.flags |= (COREDX_FACTORY_LOGGING_QOS |
    COREDX_DATA_LOGGING_QOS |
    COREDX_STATUS_LOGGING_QOS);
dr  = DDS_Subscriber_create_datareader(sub, td, &dr_qos, NULL, 0);
```

Chapter 15 CoreDX DDS Transport

The CoreDX DDS transport conforms to the Real-Time Publish-Subscribe (RTPS) Wire Protocol. This transport does not use a stand-alone transport daemon, and does not require configuration of any operating system services.

15.1 Overview

CoreDX DDS includes a modular transport infrastructure that supports configuration and customization. CoreDX DDS ships with support for UDP (on IPv4 and IPv6), TCP (IPv4, IPv6 planned), LMT (Local Machine Transport), and SERIAL. [Some platforms do not support all kinds of transports – check with Twin Oaks to determine the availability for your platform.]

Each transport implementation includes the capability to configure aspects of the transport. The set of configuration options available for each transport are described in detail in subsequent sections.

By default, CoreDX DDS will install and use the UDP transport. Alternatively, the developer can configure and install alternate or additional transports during the initialization of a DomainParticipant.

15.1.1 Transport Common API

Each transport implementation provides a set of methods to facilitate access to configuration parameters and creation of the transport; these are referred to as the **Transport Initialization API**. In addition to this API, this section also describes the DomainParticipant method to add the configured transport(s).

Using the CoreDX DDS Transport Initialization API, the developer can configure and create an instance of a transport. The API provides a mechanism to retrieve the default configuration settings. These settings can be modified by the developer as required before passing them to 'create'. The 'create' operation accepts a structure that contains all of the configuration parameters.

For example, the 'C' **Transport Initialization API** for the **UDP** transport consists of the following methods:

```
DDS_ReturnCode_t
CoreDX_UdpTransport_get_default_config(
                  CoreDX_UdpTransportConfig * config );


DDS_ReturnCode_t
CoreDX_UdpTransport_get_env_config(
                  CoreDX_UdpTransportConfig * config );


DDS_ReturnCode_t
CoreDX_UdpTransport_clear_config(
                 CoreDX_UdpTransportConfig * config );


CoreDX_Transport
*CoreDX_UdpTransport_create_transport(
                 CoreDX_UdpTransportConfig * config );
```

Each transport implementation will provide this set of Transport Initialization API methods. These methods are further described below.

15.1.1.1 Transport::get_default_config()

The get_default_config() method will initialize all fields in the provided TransportConfig structure with default values. This provides a good starting point for custom configuration modifications. If the user does not call get_default_config(), then the user is responsible for manually initializing all fields in the TransportConfig structure.

Once the TransportConfig structure has been initialized with the desired configuration values, the configuration can be passed to the create_transport() method. The transport will utilize the provide configuration values to initialize the characteristics of the transport. Once create_transport() returns, the caller can destroy or reuse the TransportConfig structure – the transport implementation does not hold any references to the provided structure.

The caller is responsible for clearing any allocated memory within the TransportConfig structure. The method clear_config() will accomplish this.

15.1.1.2 Transport::get_env_config()

This method will override values in the provided TransportConfig structure with values taken from the environment. Some transports allow the user to adjust configuration parameters by setting environment variables. This routine will query the environment and set configuration parameters based on discovered values. [Not all transports use environment variables for configuration.] In all cases, the environment variable based configuration parameters are provided as a convenience – the same result can be obtained by directly adjusting values within the TransportConfig structure.

The provided TransportConfig structure should be initialized to default values (either manually or by calling get_default_config()) prior to calling this routine.

15.1.1.3 Transport::clear_config()

This routine will clear any allocated memory within the provided TransportConfig structure. In some cases, a TransportConfig structure may contain dynamically allocated values. This memory may have been allocated by the get_default_config() or get_env_config() methods, or by the user. Each transport has a different TransportConfig structure which may or may not include parameters that have dynamic memory allocations. Clear_config() provides a mechanism to release any dynamically allocated memory within the TransportConfig structure.

15.1.1.4 Transport::create_transport()

This routine will create an instance of the transport. The returned transport instance can then be registered with a DomainParticipant. The provided TransportConfig structure will be used to configure the transport during creation. The user maintains ownership of the TransportConfig structure, and can clear or reuse it after the create_transport() call returns.

On success, the create_transport() method will return a pointer to an initialized transport instance. If there is an error during creation, NULL will be returned. Normally, the resulting transport will be passed to DomainParticipant::add_transport(). In this case, the DomainParticipant takes ownership of the transport, and the user should no longer access the transport instance. If the user does not register the transport with a DomainParticipant, then the user maintains ownership of the transport instance and is responsible for destroying it. This can be accomplished by invoking the Transport::destroy method.

15.1.1.5 DomainParticipant::add_transport()

This routine, while not strictly part of the Transport API, is used to add a configured transport to the DomainParticipant.

```
DDS_ReturnCode_t
DDS_DomainParticipant_add_transport(
                  DDS_DomainParticipant   dp,
                  CoreDX_Transport       * transport);
```

If the application does not install any transports using the add_transport() method, the DomainParticipant will automatically create and register a default UDP transport.

15.1.2 Transport Configuration

This section documents the details of each transport. This includes a discussion of the transport specific configuration parameters.

15.1.2.1 UDP Transport API

The UDP transport is the default CoreDX DDS transport. It provides the fully RTPS compliant, interoperable, DDS transport. There are many configuration options to the UDP transport.

It provides support for UNICAST and MULTICAST transmission and reception. By default, MULTICAST is enabled for both 'built-in' (discovery) data and 'user' data. For discovery configuration options over UDP, refer to Chapter 16 CoreDX DDS Discovery.

The specific configuration parameters available with the UDP transport are described in the following sections.

15.1.2.1.1 Participant Index

Participant Index is an integer number that by default is computed automatically by the RTPS implementation. It is used to distinguish DomainParticipants on the same host and in the same Domain from one another. In some cases, it is helpful to directly configure this value as it is used to compute unicast port numbers. It is an error to configure 2 or more DomainParticipants on the same host, within the same Domain, such that they have the same Participant Index value.

15.1.2.1.2 IPv4 and IPv6

The transport is configured to operate over IPv4 by default; and can be configured to support IPv6. The 'use_ipv4' and 'use_ipv6' configuration parameters provide control over which version(s) of IP will be used.

15.1.2.1.3 Interfaces

The transport, by default, will make use of all available active network interfaces. If your machine has multiple network interfaces, this may generate unnecessary network traffic on some of those networks. The transport can be configured to use a subset of available interfaces. The 'interfaces' configuration parameter can be configured with a list of interface addresses. If the list is empty, then CoreDX DDS will query the operating system to obtain a list of all available active interfaces.

15.1.2.1.4 Dynamic Interface Detection

On Operating Systems that support it, the CoreDX DDS UDP transport can detect changes to the network interfaces and adjust its configuration in response. For example, if the user brings up a new interface, CoreDX DDS will discover and utilize the new interface on the fly. This dynamic interface detection is configurable. The 'dynamic_interfaces' configuration parameter is used to enable or disable this capability.

15.1.2.1.5 RX Buffer Sizes

The UDP transport maintains buffers to handle incoming data packets. In order to preserve memory, the size and behavior of the receive buffer is configurable. By default, the receive buffer begins small and can grow dynamically as required to handle the incoming data. The initial size of the buffer is determined by the 'rx_init_buffer_size' parameter. The maximum size of the buffer is limited by the 'rx_max_buffer_size' configuration parameter. If these two values are identical, then the buffer will not dynamically resize, and all memory allocation is performed during initialization.

NOTE: If the maximum size of the RX Buffer is limited to some value smaller than that allowed by the underlying transport (in this case, UDP maximum datagram size is 64 KB), then it is possible that the transport will be forced to drop some incoming data. The size of incoming UDP datagrams is determined by the remote writing application. If the remote writer is from a CoreDX DDS DomainParticipant, then the remote peer can be configured to limit the size of transmitted packets. This configuration will enable

transmission of large data between two peers without requiring the transport to establish a large RX buffer.

15.1.2.1.6 TX Packet Size Limit

The CoreDX DDS UDP Transport transmits information in UDP datagrams. The underlying UDP transport mechanism support datagram sizes up to 64KB. In some cases, it is beneficial to limit the size of datagram put onto the network. For example, some network devices fail to handle large datagrams. The 'tx_max_packet_size' configuration parameter is used to limit the size of UDP datagram produced by the CoreDX DDS UDP Transport. CoreDX DDS will fragment the data message if it will not fit within the specified maximum size.

15.1.2.1.7 SNDBUF and RCVBUF

The UDP sockets used by the CoreDX DDS UDP Transport have an OS configured send and receive buffer. This is configurable through an Operating System provided API. In general, the OS provided default buffer sizes are appropriate; however, it is possible to override these defaults with the 'so_rcvbuf' and 'so_sndbuf' configuration parameters. For further information on these buffers, refer to the documentation for your Operating System under the topic of 'setsockopt' and SO_RCVBUF or SO_SNDBUF.

15.1.2.1.8 Multicast and Unicast

The CoreDX DDS UDP Transport supports the use of Unicast and Multicast datagrams. CoreDX DDS will use Multicast when available to minimize the number of packets written on the network. In general, this is for all communications except for:

- Heartbeat and ACK/NACK messages (RELIABILE Reliability)
- Retransmission of data packets (RELIABLE Reliability)
- Content filtered data with Writer-side filtering enabled

If Multicast is not available or desirable, then CoreDX DDS can be configured to use only Unicast transmissions. There are several configuration parameters available to tailor the use of Unicast and Multicast. In some cases, it may be useful to use Multicast for 'meta' (discovery) topics, but not for **user** topics. In some cases, it is useful to transmit multicast, but not receive it.

In order to provide full flexibility, the CoreDX DDS UDP Transport provides the following configuration parameters related to Unicast and Multicast:

Table 15-1: UDP Transport Configuration Parameters

Parameter	Description
multicast_address_v4	Specify the MULTICAST GROUP address used for all multicast communications over IPv4.
multicast_address_v6	Specify the MULTICAST GROUP address used for all multicast communications over IPv6.
multicast_ttl	Specify the MULTICAST TTL value. This defines the number of hops (routers) that multicast packets should traverse.
tx_meta_multicast	Enables the transmission of META (discovery) data over multicast.
tx_meta_unicast	Enables the transmission of META (discovery) data over unicast.
rx_user_multicast	Enables the reception of USER data over multicast.
rx_user_unicast	Enables the reception of USER data over unicast.
advertise_meta_multicast	Enables the advertisement of our ability to receive META data via multicast.
advertise_user_multicast	Enables the advertisement of our ability to receive USER data via multicast.

15.1.2.1.9 Broadcast

In some network environments, Multicast is not available or desirable. In these cases, it may be acceptable to use Broadcast as an alternative to facilitate dynamic discovery. The CoreDX DDS UDP Transport can support the broadcast of DomainParticipant discovery information. By setting

'do_meta_broadcast', the DomainParticipant Data message will be broadcast onto the local network segment.

If operating on a host or network that does not support multicast, but does support broadcast, then the following configuration may be useful:

```
advertise_meta_multicast  = 0;
advertise_user_multicast = 0;
do_meta_broadcast = 1;
```

This will prohibit remote peers from attempting multicast communication, but will support dynamic discovery via broadcast.

15.1.2.1.10 Debug

The 'debug_flags' parameter enables debug output from the UDP transport. Useful values are 2 (DATA_LOGGING) and 64 (TRANSPORT_LOGGING). The flags can be combined. See <dds/dds.h> for the full set of LOGGING flags.

15.1.2.2 UDP Transport Environment Variables

CoreDX DDS provides the ability to set many of the UDP transport configuration items through environment variables. All settings available through environment variables is also available through the Transport API.

If any transport environment variables are used to configure the UDP transport, the Transport::get_env_config() API must be called in order to apply those environment settings to a transport.

Table 15-2: UDP Transport Environment Variables

Environment Variable	Meaning	Example
COREDX_IP_ADDR	This configures the default IP address used by the UDP transport. If this value is defined in the environment, then the transport will use only the interface associated with the specified address. Legacy behavior preserved: Setting this environment variable will disable dynamic interface detection (which can be re-enabled via the Transport API if	COREDX_IP_ADDR=192.168.1.5

Environment Variable	Meaning	Example
	desired).	
COREDX_UDP_DEBUG	Sets the debug_flags parameter.	COREDX_UDP_DEBUG=64
COREDX_UDP_RX_BUFFER_SIZE	Sets the rx_max_buffer_size parameter.	
COREDX_UDP_MAX_TX_SIZE	Sets the tx_max_packet_size parameter.	
COREDX_UDP_RCVBUF	Sets the so_rcvbuf parameter.	COREDX_UDP_RCVBUF=1024
COREDX_UDP_SNDBUF	Sets the so_sndbuf parameter.	COREDX_UDP_SNDBUF=1024
COREDX_USE_MULTICAST	Sets the 'advertise_user_multicast' and 'advertise_meta_multicast' parameters.	COREDX_USE_MULTICAST=1
COREDX_MULTICAST_TTL	Sets the multicast_ttl parameter.	COREDX_MULTICAST_TTL=2
COREDX_UDP_IPV4	Sets the 'use_ipv4' parameter.	COREDX_UDP_IPV4=1
COREDX_UDP_IPV6	Sets the 'use_ipv6' parameter	COREDX_UDP_IPV6=1

15.1.2.3 TCP Transport API

The TCP transport provides support for CoreDX DDS to communicate using TCP connections. Because TCP is a connection oriented transport, there is no facility for Multicast or Broadcast. Without Multicast or Broadcast, the TCP transport does not provide any facilities for fully Dynamic Discovery.

The current version of the TCP transport supports only IPv4. Support for IPv6 is planned for a subsequent release.

The specific configuration parameters available with the TCP transport are described in the following sections.

15.1.2.3.1 Participant Index

Participant Index is an integer number that by default is computed automatically by the RTPS implementation. It is used to distinguish DomainParticipants on the same host and in the same Domain from one another. In some cases, it is helpful to directly configure this value as it is *used to compute unicast port numbers. It is an error to configure 2 or more DomainParticipants on the same host, within the same Domain, such that they have the same Participant Index value.

15.1.2.3.2 Interfaces

The transport, by default, will make use of all available active network interfaces. If your machine has multiple network interfaces, this may generate unnecessary network traffic on some of those networks. The transport can be configured to use a subset of available interfaces. The 'interfaces' configuration parameter can be configured with a list of interface addresses. If the list is empty, then CoreDX DDS will query the operating system to obtain a list of all available active interfaces.

15.1.2.3.3 Dynamic Interface Detection

On Operating Systems that support it, the CoreDX DDS TCP transport can detect changes to the network interfaces and adjust its configuration in response. For example, if the user brings up a new interface, CoreDX DDS will discover and utilize the new interface on the fly. This dynamic interface detection is configurable. The 'dynamic_interfaces' configuration parameter is used to enable or disable this capability.

15.1.2.3.4 TX Packet Size Limit

The CoreDX DDS TCP Transport transmits information in RTPS Messages. In some cases, it is beneficial to limit the size of RTPS Messages put onto the network. For example, some network devices fail to handle large packets. The 'tx_max_packet_size' configuration parameter is used to limit the size of RTPS Messages produced by the CoreDX DDS TCP Transport. CoreDX DDS will fragment the data message into multiple smaller messages, if it will not fit within the specified maximum size.

15.1.2.3.5 Debug

The 'debug_flags' parameter enables debug output from the UDP transport. Useful values are 2 (DATA_LOGGING) and 64 (TRANSPORT_LOGGING). The flags can be combined. See <dds/dds.h> for the full set of LOGGING flags.

15.1.2.4 TCP Transport Environment Variables

CoreDX DDS provides the ability to set the TCP transport configuration items through environment variables. The Transport::get_env_config() API must be called in order to apply those environment settings to a transport.

Table 15-3: TCP Transport Environment Variables

Environment Variable	Meaning	Example
COREDX_IP_ADDR	This configures the default IP address used by the TCP transport. If this value is defined in the environment, then the transport will use only the interface associated with the specified address. Legacy behavior preserved: Setting this environment variable will disable dynamic interface detection	COREDX_IP_ADDR=192.168.1.5
COREDX_TCP_DEBUG	Enables debug output from the TCP transport. Useful values are 2 (DATA_LOGGING) and 64 (TRANSPORT_LOGGING). The flags can be combined. See <dds/dds.h> for the full set of LOGGING flags.	COREDX_TCP_DEBUG=64

15.1.2.5 LMT Transport API

The LMT (Local Machine Transport) provides support for optimized 'on host' communication. The transport enables DomainParticipants that are co-located on the same host to communicate with lower overhead and latencies than provided by the default UDP transport.

The LMT transport is currently provided on the Linux Operating System. [A Windows port of LMT is planned.]

NOTE: LMT is not implemented using shared memory. As a result, the safety provided by separate program address spaces is maintained.

The specific configuration parameters available with the UDP transport are described in the following sections.

15.1.2.5.1 SNDBUF and RCVBUF

The UDP sockets used by the CoreDX DDS UDP Transport have an OS configured send and receive buffer. This is configurable through an Operating System provided API. In general, the OS provided default buffer sizes are appropriate; however, it is possible to override these defaults with the 'so_rcvbuf' and 'so_sndbuf' configuration parameters. For further information on these buffers, refer to the documentation for your Operating System under the topic of 'setsockopt' and SO_RCVBUF or SO_SNDBUF.

15.1.2.5.2 TX Packet Size Limit

In some cases, it is beneficial to limit the size of packet written by the transport. The 'max_tx_size' configuration parameter is used to limit the size of packet produced by the CoreDX DDS LMT Transport. CoreDX DDS will fragment the data message if it will not fit within the specified maximum size. Refer to the blah section for additional information on buffer sizing and fragmentation.

15.1.2.5.3 RX Buffer Size

The LMT transport maintains buffers to handle incoming data packets. In order to preserve memory, the size and behavior of the receive buffer is configurable. By default, the receive buffer begins small and can grow dynamically as required to handle the incoming data. The initial size of the buffer is set at initialization to be just large enough to handle an RTPS Message Header. The maximum size of the buffer is limited by the 'max_rx_buf_size' configuration parameter.

15.1.2.5.4 Debug

The 'debug_flags' parameter enables debug output from the LMT transport. Useful values are 2 (DATA_LOGGING) and 64 (TRANSPORT_LOGGING). The flags can be combined. See <dds/dds.h> for the full set of LOGGING flags.

15.1.2.6 LMT Transport Environment Variables

CoreDX DDS provides the ability to set many of the LMT transport configuration items through environment variables. All settings available through environment variables is also available through the Transport API.

If any transport environment variables are used to configure the LMT transport, the Transport::get_env_config() API must be called in order to apply those environment settings to a transport.

Table 15-4: LMT Transport Environment Variables

Environment Variable	Meaning	Example
COREDX_LMT_DEBUG	Sets the 'debug_flags' parameter.	COREDX_LMT_DEBUG=64
COREDX_LMT_RCVBUF	Sets the 'rcvbuf' parameter.	
COREDX_LMT_SNDBUF	Sets the 'sndbuf' parameter.	
COREDX_LMT_MAX_TX_SIZE	Sets the 'max_tx_size' parameter.	
COREDX_LMT_MAX_RX_BUF_SIZE	Sets the 'max_rx_buf_size' parameter.	

15.1.2.7 UDS Transport

The UDS transport provides the facility for serial or other stream based transports. UDS is supported by a helper program that initializes the serial port, and provides a multi-participant access to the single shared resource. Because of the diverse nature of serial and other related hardware devices, please contact Twin Oaks Computing for assistance in adapting this transport technology to your platform.

Chapter 16 CoreDX DDS Discovery

16.1 Overview of CoreDX DDS Discovery

The automatic discovery process is one of the more powerful and useful features of CoreDX DDS. Automatic discovery of entities allows CoreDX DDS applications to publish and subscribe to data without needing to configure the endpoint(s) they talk to. Whether these endpoints are on the same machine, or across the room, CoreDX DDS applications do not need any knowledge of the other applications they will be communicating with.

The Standard (peer-to-peer, dynamic) Discovery process in CoreDX DDS is encapsulated in every CoreDX DDS application, and does not require any additional daemons or services. Each CoreDX DDS application performs the discovery process, including announcing the presence of its DDS Entities, listening for other DDS Entities, and looking for matches between its own DDS Entities and those discovered. The standard discovery mechanism is interoperable between DDS implementations.

The automatic discovery process includes the following steps:

1. Discovering DomainParticipants
2. Discovering DataReaders and DataWriters within those discovered Participants
3. *Matching* those discovered DataReaders and DataWriters with local DataReaders and DataWriters

Despite the many benefits of the Standard Discovery mechanism, it does have some drawbacks for certain system architectures. For example, where security requirements prevent dynamic discovery, or where scalability requirements need an alternative solution to the standard peer-to-peer discovery. CoreDX DDS provides several alternatives for configuring the discovery process.

16.2 Discovering DomainParticipants

The first step in the automatic discovery process is to discover remote DomainParticipants. Each DomainParticipant will periodically announce its existence (including how it can be reached directly to learn about containing DataReader and DataWriter Entities) by writing a

SPDPdiscoveredParticipantData message to the multicast address specified by the DDS standards. Each DomainParticipant will also listen on that same multicast address to learn about other DomainParticipants.

After a DomainParticipant has been discovered, it will be considered 'alive' as long as its SPDPdiscoveredParticipantData messages continue to be received. If enough time expires without receiving a SPDPdiscoveredParticipantData message from a DomainParticipant, that DomainParticipant is no longer considered 'alive'.

16.2.1 Configuring Participant Discovery

The DDS standards specify default durations for how often SPDPdiscoveredParticipantData messages should be sent, and how much time should expire before a DomainParticipant should be considered 'not alive'.

While these default durations work well for most network environments, they may not work for all environments. For example, networks with very long latencies, or extremely bandwidth constrained networks may need to tailor the timing of discovery messages.

CoreDX DDS allows the application to configure timing of DomainParticipant discovery by using the CoreDX_DiscoveryQosPolicy, as described below.

QoS Policy	Default Value	Description
CoreDX_DiscoveryQosPolicy		
dpd_push_period (duration_t)	5 seconds	Configure the amount of time between sending of SPDPdiscoveredParticipantData messages.
dpd_lease_duration (duration_t)	120 seconds	Configure the amount of time the can expire without receiving a SPDPdiscoveredParticipantData message before we consider a remote Participant be 'not alive'.

16.3 Matching DataReaders and DataWriters

The matching of DataReaders and DataWriters is a sophisticated process that ensures the subscribers of data are matched appropriately with producers of data. This careful matching helps to reduce programming errors, in addition to reducing unnecessary storage and network communications usage.

The following three conditions are checked between DataReaders and DataWriters, and all must be met in order to create a *match*:

1. The Topic Names must match

   The name used when creating a Toipc with DomainParticipant::create_topic() is communicated to peer DomainParticipants during discovery. The name of the Topic used by a DataReader must match the name of the Topic used by a DataWriter.

2. The Type Names (and type definition, if available) must match

   The name used when registering a data type with TypeSupport::register_type() is communicated to peer DomainParticipants during discovery. The name of the registered type associated with the Topic used by a DataReader must match the name of the registered type associated with the Topic used by a DataWriter.

   Further type checking is performed when the type definition (called the *typecode*) is available. The *typecode* is an encoding of the type definition as defined in the DDL, and provides more accurate information about the data type actually being published by DataWriters and expected to be received by DataReaders. Using typecodes in Entity matching provides an additional level of type safety.

   By default, CoreDX DDS applications exchange typecode information during discovery.

3. The QoS policies must be compatible

   The QoS policy defined with DataReaders and DataWriters is communicated to peer DomainParticipants during discovery. Once the Topics and Types have been verified as matching between a DataReader and DataWriter, the QoS policies are checked for compatibility. Refer to

section Part 3:12.1 *QoS Compatibility* for additional information on QoS compatibility matching.

16.3.1 Configuring Entity Matching

The typecode is not required for DDS discovery. By default, CoreDX DDS will send and use the typecode for Entity matching, but the application can configure this behavior.

QoS Policy	Default Value	Description
CoreDX_RTPSReaderQosPolicy		
send_typecode (unsigned char)	1 (true)	Configure this DataReader to send (or not send) the typecode for the Topic it is subscribing to.
CoreDX_RTPSWriterQosPolicy		
send_typecode (unsigned char)	1 (true)	Configure this DataWriter to send (or not send) the typecode for the Topic it is publishing.

There are several possible reasons an application may need to disable the sending of typecodes (and therefore, removing the capability to use them for Entity matching during discovery).

Because type encoding was not well specified in the earlier DDS standards, it is possible that DDS implementations are not interoperable in matching all possible typecodes.

In addition, typecodes require system resources: generated code uses additional static or FLASH memory, sending and receiving typecodes uses additional dynamic or RAM memory and network resources. This is especially true of large type definitions. In extreme cases, the typecode may be too large to send over the available transport (this is especially true for low bandwidth transports).

16.4 Static Discovery

The CoreDX DDS middleware supports dynamic discovery by default. This allows DomainParticipants to discover other Participants in the same domain. Once DomainParticipants discover each other, then the contained DataReaders and Writers are dynamically matched.

The standard dynamic discovery protocol is based on UDP MULTICAST, and works effectively in a local area network, and can be configured to work through routers and other networking devices. However, there are some situations in which UDP MULTICAST is not desirable or possible, or where Dynamic Discovery is not suitable for the application.

In order to address these situations, CoreDX DDS provides QoS support for defining remote peer DomainParticipants. An extension QoS policy is added to the DomainParticipant QoS policies. This policy, peer_participants, identifies a list of remote DomainParticipants with which this participant should communicate. This avoids the initial participant discovery process, and initiates direct communication between the identified participants.

This QoS policy can be updated on the fly, after the DomainParticipant is enabled, and can support an application controlled discovery mechanism. Table 16-1 shows a C language example of setting the 'peer_participant' QoS policy.

Table 16-1: Code Example of peer_participants QoS

Example code to pre-define Peer Participants

```
DDS_DomainParticipant    dp;
DDS_ReturnCode_t         retval;
DDS_DomainParticipantQos dp_qos;
CoreDX_ParticipantLocator pl;

DDS_DomainParticipantFactory_get_default_participant_qos(&dp_qos);

/* add two 'a-priori' configured peer locators */
/* two different participant id's and two different IP addrs */
pl.participant_id = 0;
pl.participant_locator.kind = COREDX_UDPV4_LOCATOR_KIND_QOS;
memset(pl.participant_locator.addr, 0, 16);
pl.participant_locator.addr[12] = 192;
pl.participant_locator.addr[13] = 168;
pl.participant_locator.addr[14] = 1;
pl.participant_locator.addr[15] = 12;
seq_add(&dp_qos.peer_participants.value, &pl);
```

```
pl.participant_id = 1;
pl.participant_locator.kind = COREDX_UDPV4_LOCATOR_KIND_QOS;
memset(pl.participant_locator.addr, 0, 16);
pl.participant_locator.addr[12] = 192;
pl.participant_locator.addr[13] = 168;
pl.participant_locator.addr[14] = 1;
pl.participant_locator.addr[15] = 22;
seq_add(&dp_qos.peer_participants.value, &pl);

dp = DDS_DomainParticipantFactory_create_participant( 0,
                                                      &dp_qos,
                                                      NULL, 0);
```

Add Java example, note C# missing API?

16.5 Centralized Discovery

16.5.1 Overview

The Standard (peer-to-peer) Discovery process in CoreDX DDS is encapsulated in every CoreDX DDS application, and does not require any additional daemons or services. Each CoreDX DDS application performs the discovery process, including announcing the presence of its DDS Entities, listening for other DDS Entities, and looking for matches between its own DDS Entities and those discovered. The standard discovery mechanism is interoperable between DDS implementations.

This Standard Discovery is depicted in the below diagram.

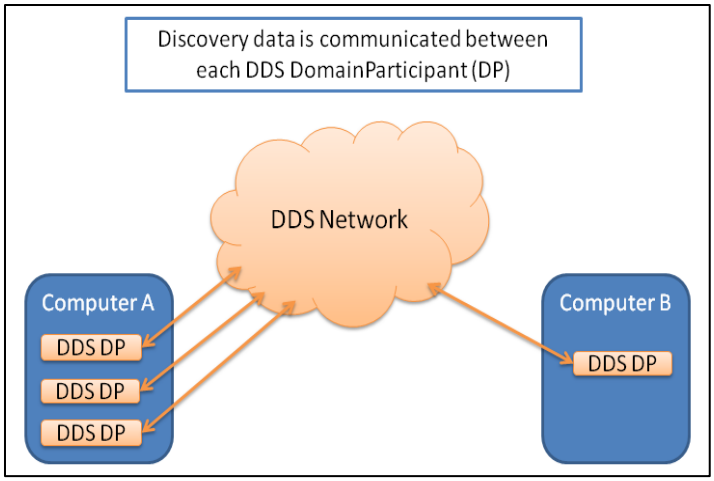

Figure 17: Standard Discovery (peer-to-peer) architecture

Despite the many benefits of the Standard Discovery mechanism, it does have some drawbacks for certain system architectures. In particular, Standard Discovery may not scale well to large DDS domains. In DDS domains with large numbers of DDS entities (Participants, Readers or Writers), the Standard Discovery mechanism can require large amounts of memory as every Participant discovers all other entities in the system. In many case, this 'world view' of the DDS domain is wasteful. Often, a Participant is required to communicate with only a small sub-set of the entire DDS network.

To address the scalability issues of Standard Discovery, CoreDX DDS supports a specialized discovery mechanism called **Centralized Discovery**. CoreDX Centralized Discovery performs the work of discovering all DDS Entities and appropriately communicating those entities to participants based on 'need to know'. The Centralized Discovery mechanism can scale to very large DDS domains, without the explosion of memory allocation found in Standard Discovery.

Further, Centralized Discovery is designed to be interoperable with Standard Discovery. This means that a DDS domain may combine both discovery mechanisms as necessary: some Domain Participants can use Standard Discovery while others use Centralized Discovery.

This Centralized Discovery is depicted in the below diagram.

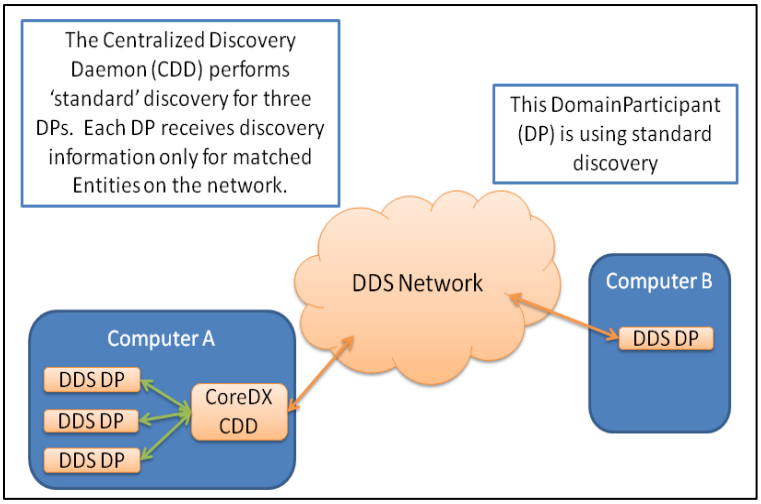

Figure 18: Centralized Discovery architecture

16.5.2 Memory Utilization and Scalability

With Standard Discovery, each DomainParticipant learns and remembers every active DomainParticipant, DataReader, and DataWriter in the DDS Domain. As the number of DDS Entities in the Domain grows, so does the amount of discovery information stored in each DomainParticipant.

For systems that contain many DDS Entities, it may be desirable to reduce the number of copies of this maintained discovery information. This is one benefit of Centralized Discovery. The discovery information about all DDS Entities in a DDS Domain is stored in a centralized location, reducing the overall memory utilization in the system.

The Centralized Discovery Daemon determines the potential Reader/Writer matches for all its connected DomainParticipants. DomainParticipants learn only about potential matches from the Centralized Discovery Daemon.

A CoreDX Centralized Discovery Daemon must reside on the same machine as DomainParticipants that are configured to use Centralized Discovery. Therefore, the greatest benefits for memory reduction are seen when:

1. There are many DDS Entities on one machine that can use Centralized Discovery, and
2. For each DomainParticipant, a small percentage of the DDS Entities in the DDS domain match with its own DDS Entities.

Note that the selection of 'discovery' mechanism affects only the exchange of discovery information – application data is not affected. Application data is always exchanged peer-to-peer, even when using Centralized Discovery.

16.5.3 Network Utilization and Discovery

With Standard Discovery, each DomainParticipant communicates with every active DomainParticipant in the DDS Domain. As the number of DDS DomainParticipants in the Domain grows, so does the amount of network traffic generated to communicate with each peer DomainParticipant.

For systems that contain many DomainParticipants, and at least some of these DomainParticipants are co-located on the same computer, it may be desirable to reduce the number messages generated on the network. This is an additional benefit of Centralized Discovery. The DomainParticipants co-located on a computer will communicate with their location Centralized Discovery Daemon, and only the Centralized Discovery Daemon will communicate off-box, reducing the amount of discovery network traffic.

16.5.4 Configuring Centralized Discovery

Configuring CoreDX DDS discovery happens at the DomainParticipant using a QoS policy. A DomainParticipant can be configured to use a certain discovery mechanism at creation time through a QoS policy.

By default, CoreDX DDS uses Standard Discovery (PEER). To use Centralized Discovery, change the DomainParticipant CoreDX_Discovery_Qos_Policy to specify centralized discovery. The DomainParticipant QoS policy for configuring discovery (and built-in entities) is described below.

CoreDX Centralized Discovery is compliant with the OMG's RTPS Wire Protocol standard, and is therefore interoperable with other DDS implementations. Since DomainParticipants using Centralized Discovery can communication with DomainParticipants using standard discovery, a mix of discovery types can be configured in the same DDS network.

QoS Policy	Default Value	Description
CoreDX_DiscoveryQosPolicy		
discovery_kind	DDS_PEER_	Configure this DomainParticipant

QoS Policy	Default Value	Description
(DiscoveryQosPolicyDiscoveryKind)	DISCOVERY_QOS	to use standard (PEER) discovery or centralized discovery. Possible values are: DDS_PEER_DISCOVERY_QOS and DDS_CENTRAL_DISCOVERY_QOS

16.5.5 Deploying Centralized Discovery

A CoreDX Centralized Discovery Daemon must be deployed on each computer that is hosting a DomainParticipant configured to use Centralized Discovery. There should be only one CoreDX Centralized Discovery Daemon running on a computer. Computers that are not hosting DomainParticipants configured to use Centralized Discovery do not need a CoreDX Centralized Discovery Daemon.

An example deployment using Centralized Discovery is shown below.

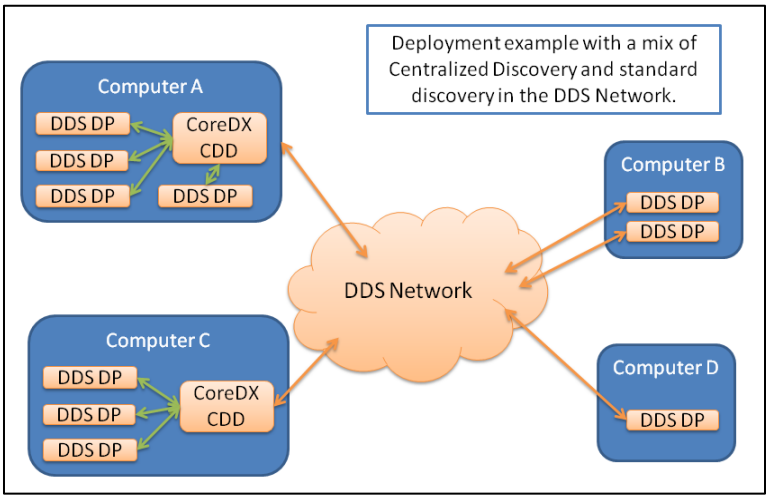

Figure 19: Example Centralized Discovery deployment

16.6 Wait for Discovery

The automatic and dynamic discovery process is defined by the DDS standards and employed by CoreDX DDS. While the discovery algorithms are efficient, the dynamic nature of discovery means it is impossible to

determine the amount of time required for discovery to complete, even when all entities are located within one DomainParticipant.

For example, consider an application with the following execution flow (and no pauses or gaps between steps):

1. Create DomainParticipant
2. Register data types and create Topics
3. Create Subscribers and DataReaders
4. Create Publishers and DataWriters
5. Write data

Discovery and matching of these local DataReaders and DataWriters may not complete before the application writes data. If the discovery and matching is not yet complete, data will not be received by the DataReaders (since they are not yet known to exist by the DataWriter).

To address this problem, CoreDX DDS provides an API on DomainParticipants to wait for built-in acknowledgements:

```
DomainParticipant::builtin_wait_for_acknowledgments(
Duration_t *max_wait)
```

While a DomainParticipant configured to use dynamic discovery has no way to know how many, if any remote DDS Entities may be discovered, this API will block the application until all DataReaders and DataWriters within known DomainParticipants have been discovered and matched as appropriate (or until the max_wait duration has expired).

16.7 Discovery and Deterministic Machines

CoreDX DDS discovery adheres to the DDS standards and is interoperable with other DDS implementations. Part of this interoperable discovery process is the assignment of Globally Unique Identifiers (GUIDs) to RTPS Entities that are advertised and may be discovered by other RTPS Entities, including Participants, Readers, and Writers.

The Participant GUID is important for dynamic discovery. Each Participant will generate a unique GUID and include it in the discovery messages it publishes. When a new discovery advertisement message is received, a Participant can use the GUID to determine if this is new Participant, or one that it has already discovered. Newly discovered participants will

participate in additional data exchange to share QoS policy settings and information about existing DataReaders and DataWriters.

The discovery process allows an application's Participant to go away (by normal or abnormal exit, or machine restart), restart, and seamlessly re-join the existing DDS network as a *new* Participant. This works only if the restarting application's Participants are assigned a *unique* GUID.

According to the standard, the Participant GUID is created using the following data:

- IPv4 address of the computer hosting the Participant
- Process ID of this Participant's application
- One-up counter for each Participant within this application
- Entity Kind (fixed identifier for Participant)

For many computer systems, this algorithm does generate unique GUID's for Participants, even after machine restarts. However, applications running on deterministic Operating Systems, such as VxWorks, may always start with the same process ID, resulting in a Participant always having the same GUID. This can cause a problem when a Participant attempts to re-join DDS communications using the same GUID it had previously. Remote Participants on the DDS network will consider this Participant an already-discovered Participant, and will not participate in the necessary data exchange to share QoS policy settings and existing DataReaders and DataWriters.

To address this problem with deterministic systems, CoreDX DDS provides an additional discovery QoS policy setting for applications to use their own algorithm to set the Process ID portion of the GUID. When used, this should be a number that uniquely identifies an application on a computer, and will not be the same after a machine restart. This might be a one-up counter that is written to persistent storage (disk, writable FLASH memory) or another application defined algorithm.

QoS Policy	Default Value	Description
CoreDX_DiscoveryQosPolicy		
guid_pid (DDS_BUILTIN_TOPIC_KEY_TYP	0	A value of '0' will use the default: the application process ID (PID).

QoS Policy	Default Value	Description
E_NATIVE)		Values other than '0' will be used in place of the PID in constructing the GUID.

Chapter 17 Configuring Reliability Protocol

17.1 Reliability Protocol

The CoreDX DDS Reliability protocol addresses dropped packets, out of order samples, communication disconnects, and application re-starts to ensure delivery of published data to the intended recipients. This protocol is supported by the RTPS protocol and the DataReader and DataWriter Data Caches (see Chapter 10.5 for a full discussion on the Data Caches).

This reliability protocol is light-weight and minimizes latency. Dropped packets are quickly detected and repaired. CoreDX DDS provides tunable parameters for configuring the reliability protocol to allow the application developer to achieve an optimal balance of overhead and timely data retransmission.

17.1.1 Cache Management

The reliable protocol effects more than the handshaking between DataWriters and DataReaders. The Data Cache also plays an important role in reliable communications.

On the DataWriter, the Data Cache contains samples that have not been acknowledged by all reliable subscriptions. [Data may be kept longer based on the Durability QoS configuration.] The data cache size is controlled by the History and the Resource Limits QoS policies. If the data cache becomes full, the History QoS policy kind comes into play. With a History kind of KEEP_ALL, the write() call will block until there is space, or until the max_blocking_time has elapsed. With a History kind of KEEP_LAST, the oldest sample will be removed from the cache to make room for a new sample.

On the DataReader, the Data Cache contains samples in order, with respect to the source DataWriter. If samples are lost, the data cache may contain out of order samples. These samples are stored in a "forward cache" and are not available to the application until all missing samples are received. This design minimizes retransmissions of data samples.

17.1.2 Heartbeats, ACKs, and NACKs

The reliability protocol relies on *Heartbeats* from the DataWriter and *ACK/NACK responses* from the DataReader. *Heartbeat* messages tell the DataReaders the data that is currently available (that has been sent) at the DataWriter. Positive *ACK* and negative *NACK Responses* are sent in response to a *Heartbeat* and confirm the DataReader has received one or more samples, and possibly requests one or more samples to be retransmitted.

Figure 17-1 shows a simple example of the network traffic (including *Heartbeat* and *ACK Responses*) when there are no dropped samples. Notice that *Heartbeats* can be sent in combination with data samples, reducing network overhead.

Figure 17-1: Example DDS Usage

In this example, the DataReader sends an ACK in response to each of the 2 Heartbeats received from the DataWriter. In the first ACK response, the DataReader confirms receipt of all available samples up to sample #3. In the second ACK response, the DataReader confirms receipt of all available samples up to sample #6.

Figure 17-2 shows a similar example, except one data sample has been lost, and must be retransmitted.

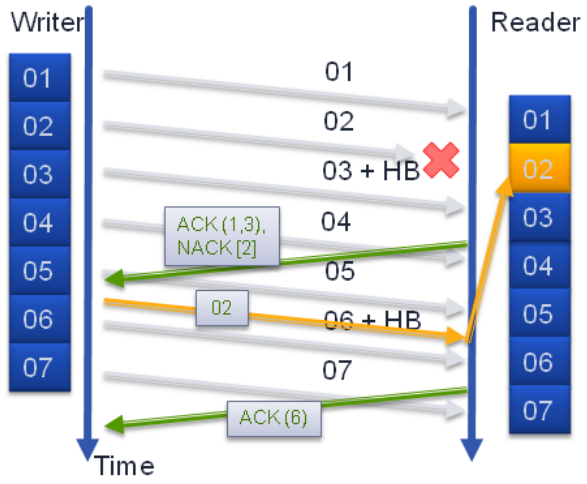

Figure 17-2: Example DDS Usage

In this example, the DataReader sends an ACK/NACK in response to the first Heartbeat from the DataWriter. The DataReader can acknowledge samples #1 and #3, but not sample #2. When the DataWriter receives the NACK, it will retransmit sample #2, sending it directly to this DataReader who needs it. In response to the next Heartbeat from the DataWriter, the DataReader can now acknowledge all data samples through #6.

17.1.3 Unresponsive DataReaders

DataReaders that do not respond to Heartbeats with ACK/NACK messages in a timely manner pose a unique challenge to the standardized DDS Reliability protocol. A DataWriter with certain QoS policy settings will keep each published sample in its data cache until it is acknowledged by all matched Reliable DataReaders. If a DataReader becomes *unresponsive* (that is, the DataWriter is no longer receiving ACK/NACK messages from this DataReader), the DataWriter is unable to purge samples from its cache. After some period of time, the data cache may become full, or available memory will be used to store these un-acknowledged samples. At this time the application may be unable to write additional data (exact behavior will depend on other QoS policy configuration), and the other, responsive DataReaders will not receive any additional data, all because of 1 *unresponsive* DataReader.

To address this challenge, CoreDX DDS will degrade *unresponsive* DataReaders to a Reliability configuration that not quite Reliable. The DataWriter will continue to send Heartbeats to this DataReader (in the hope

that it will eventually respond with ACK/NACK messages), but will no longer expect (or wait for) samples to be acknowledged by this DataReader. In the above example, the DataWriter will remove samples from its data cache once they are acknowledged by all responsive DataReaders, allowing data communications to continue without interruption. [Note that the unresponsive DataReader may miss samples.]

When a DataReader is marked *unresponsive* is configurable by the application through the ack_deadline parameter (described below).

17.2 Reliability QoS Configuration

Some applications need the ability to tailor the reliability protocol to achieve an optimal balance of overhead and timely data retransmission.

CoreDX DDS provides additional QoS policies to allow tailoring of the RELIABLE handshaking protocol: the DataWriter_RTPS_WriterQoSPolicy and the DataReader_RTPS_ReaderQoSPolicy. These QoS policies allow the application to tailor when and how frequently CoreDX DDS sends heartbeats, NACKs, and related responses. Configuring these items can balance latency and overhead in RELIABLE communications, and help avoid packet storms.

These additional QoS policies are described below.

Table 17-1: CoreDX DDS RTPS_Protocol QoS Policy

QoS Policy	Default Value	Description
DataWriter_RTPSWriterQoSPolicy		
heartbeat_period (Duration_t)	2 ms	The duration between Heartbeats sent by the DataWriter.
nack_response_delay (Duration_t)	1 us	The delay between the receipt of a NACK Response and the DataWriter's retransmission of data.

QoS Policy	Default Value	Description
nack_suppress_delay (Duration_t)	0 ns	Not yet used by CoreDX DDS
ack_deadline (Duration_t)	INFINITE	The amount of time the DataWriter should wait for an ACK/NACK message from a DataReader before it is considered *unresponsive*.
DataReader_RTPSReaderQoSPolicy		
heartbeat_response_delay (Duration_t)	500 us	The delay between the receipt of a Heartbeat and the DataReader's ACK/NACK Response.
send_initial_nack	1 (true)	This setting is applicable only to Reliable DataReaders. When set to non-zero (true), the DataReader will send a "NACK" message to each newly discovered DataWriter, jumpstarting the handshaking process to receive any data the DataWriter has to publish. There are a couple of possible reasons to disable this 'jumpstart', including performance (with lots of DataReaders matching with one DataWriter at the same time), or interoperability (in our testing to date, one DDS vendor did not support this option).
CoreDX_DiscoveryQosPolicy		
heartbeat_period (Duration_t)	10 ms	For built-in writers, the delay between the receipt of a NACK Response and the DataWriter's retransmission of data.

QoS Policy	Default Value	Description
nack_response_delay (Duration_t)	1 us	For built-in writers, the delay between the receipt of a NACK Response and the DataWriter's retransmission of data.
nack_suppress_delay (Duration_t)	0 sec	No yet used by CoreDX DDS
heartbeat_response_delay (Duration_t)	0 sec	For built-in readers, the delay between the receipt of a Heartbeat and the DataReader's ACK/NACK Response.
send_initial_nack	1 (true)	When set to non-zero (true), the built-in DataReaders will send a "NACK" message to each newly discovered built-in DataWriter, jumpstarting the handshaking process to receive any discovery data the DataWriter has to publish. This may need to be disabled to support interoperability (in our testing to date, one DDS vendor did not support this option).

Chapter 18 Dynamic Types

18.1 Overview

The original DDS standards ware designed around the assumption that types associated with Topics are known at compile time. While this architecture provides better communication performance (throughput and latency) and type safety, static data typing makes it difficult to dynamically define Topics at run time.

The addition of Dynamic Types to the CoreDX DDS baseline allows greater flexibility to application developers. Developers can define their data types at compile time or discover data types at run time. Once a data type is discovered, the application can dynamically create DataReaders to receive Topic data and use introspection to parse and process received data. DataWriters can also be created to write the discovered data type.

The dynamic type technology can also benefit applications that will be deployed to very space constrained environments. Using Dynamic Types can help reduce the code size by reducing or eliminating the type specific code generated from DDL.

CoreDX DDS Dynamic Types are interoperable with other RTPS compliant DDS implementations, so CoreDX DDS applications can discover type information from other vendor's DataWriters, and other vendor's DDS applications can discover type information from CoreDX DDS DataWriters. This interoperability is a result of the ***Type Codes*** that are written on the wire. The Type Codes contain all the information on the data type associated with a Topic, and is contained within the DataWriter and DataReader discovery information. When a DDS application discovers a new DataWriter or DataReader, it can obtain the type information. This type information can be used to create a DynamicType_DataReader to read data or a DynamicType_DataWriter to write data.

18.2 Dynamic Data Type Entities

Standard user defined data types are defined by the application developer in an IDL-like syntax and passed to the CoreDX DDS compiler. The CoreDX DDS compiler creates type-specific Entities, including a Type, TypeSupport, DataReader and DataWriter (see Section 11.11: Generated Code for

additional information on non-dynamic data types). These same Entities exist for Dynamic Data Types and are described in the following table.

Table 18-1: Dynamic Data Type Entities

Dynamic Entity (C naming convention used)	Description
DDS_DynamicType	The Type definition for a DynamicType contains functions for determining the data type and accessing the data.
DDS_DynamicType_TypeSupport	The TypeSupport for a DynamicType behaves similarly to non-dynamic type specific TypeSupport: it contains a method to register this type with the CoreDX DDS infrastructure.
DDS_DynamicType_DataReader	The DataReader for a DynamicType is used the same as any DataReader and contains the same functions as non-dynamic type specific DataReaders.
DDS_DynamicType_DataWriter	The DataWriter for a DynamicType is used the same as any DataWriter and contains the same functions as non-dynamic type specific DataWriters.

DDS_DynamicType (**DDS::DynamicType** for C++ and Java[1]). This data type contains all the specialized type support code for reading and writing on the DDS network, as if it had been generated using the CoreDX DDL compiler.

A dynamic type includes additional helper functions to determine the data and metadata of a DDS_DynamicType instance. These functions are described below.

[1] CoreDX DDS currently supports Dynamic Types for the C, C++ and Java language bindings. C# support will be available in a future release.

For C developers, all function names listed below are prefixed with "DDS_". All functions take, as their first argument, the DDS_DynamicType instance to work on.

For C++ and Java developers, all function names listed below are in the DDS name space.

18.3 Using Dynamic Types

Using Dynamic Types for publishing data or receiving data involves several steps.

1. Define the data type [This can be done *manually* or using a TypeSupport *helper function*.]
2. Register the Dynamic Type

For Subscribing Applications:

3. Create a Dynamic Type DataReader
4. Call a variant of read() or take() on the Dynamic Type DataReader
5. Access data in an instance

For Publishing Applications:

6. Create a Dynamic Type DataWriter
7. Initialize an instance
8. Call a variant of write() on the Dynamic Type DataWriter

18.3.1 Defining a Dynamic Type

18.3.1.1 Basic Types

The fundamental data types (short, long, char, etc) are represented by creating a DynamicType element to represent the data. For example, in C, the function **DDS_DynamicType_alloc_basic(DDS_TYPECODE_LONG)** will create a DynamicType object that represents a 'long' data element. In C++ or Java, simply allocate an object of type DDS::LongDynamicType() to achieve the same thing; for example, **new LongDynamicType()**.

18.3.1.2 Structures

To define a structure, the number of fields must be configured, and then each field must be created and assigned to the structure.

18.3.1.3 Arrays

To define an array dynamic type, the element type and max length must be configured. Use the set_max_length() and set_element_type() methods to define an array.

18.3.1.4 Sequences

To define a sequence data type, the element type must be configured. For bounded sequences, the max length must also be configured. (The default max length is 0, which indicates an unbounded sequence).

18.3.1.5 Strings

Bounded strings require the max length to be configured.

18.3.1.6 Unions

Unions are similar to structures in that it has several members defined. However, a union instance can hold only a single one of those elements at any one time. The current data element is identified by the 'discriminator'. The DynamicType Union interface includes methods to define the full set of data fields, the set of 'labels' (discriminator values) associated with each field, and the 'default' field.

Here is a C++ example that will 'construct' a Union DynamicType instance. The union has three fields ((0) a char, (1) a long, and (2) a fixed string). The discriminator is an octet (byte),field number 2 (starting from zero) is the 'default' field.

```
UnionDynamicType *udt = new UnionDynamicType();
dt1 = new OctetDynamicType();
udt->set_discriminator(dt1);
udt->set_num_fields(3);
udt->set_default_field(2);

dt1 = new CharDynamicType();
udt->set_field(0, "a_char", dt1);
udt->set_field_num_labels(0, 1);
udt->set_field_label(0, 0, 1);

dt1 = new LongDynamicType();
udt->set_field(1, "a_long", dt1);
udt->set_field_num_labels(1, 1);
udt->set_field_label(1, 0, 2);
```

```
dt1 = new StringDynamicType();
((StringDynamicType*)dt1)->set_max_length(16);
udt->set_field(2, "a_fixedstring", dt1);
udt->set_field_num_labels(2, 1);
udt->set_field_label(2, 0, 3);
```

18.3.2 Initializing a Dynamic Type Instance

CoreDX DDS provides a number of functions to create a dynamic data type and initialize the data in a dynamic type instance. These functions are useful for publishing applications. These applications can publish data to a discovered Topic where the data type was not known at compile time.

18.3.2.1 Basic Types

The DynamicType API includes routines to set the data element value of a 'basic' data field. For example, in C++:

```
LongDynamicType * dt = new LongDynamicType();
dt->set_long(123);
```

18.3.2.2 Structures

A structure is initialized by initializing the data values of each of its fields. The data values can be initialized prior to adding the fields to the structure, or after. Further, the values of contained fields can be changed at any time during the lifetime of the structure object.

18.3.2.3 Arrays

The application must call set_length() to allocate space for array elements, (len==max_len), and set_element() for each element 0..len-1.

18.3.2.4 Sequences

The application must call set_length() to allocate space for elements, (len <= max_len), and set_element() for each element 0..len-1.

18.3.2.5 Strings

The application must call set_string(). CoreDX DDS will truncate the string to max_len if necessary.

18.3.2.6 Unions

In order to set the data value of a union, it is necessary to first set the value of the union discriminator. This value is used to determine which of the union's data members are currently 'selected'.

Once the data type is created, here is a C++ example of how to assign values to the union:

```
OctetDynamicType * discrim =
    (OctetDynamicType*)udt->get_discriminator();
discrim->set_octet(1);
CharDynamicType * cdt =
    (CharDynamicType *)udt->get_field(0);
cdt->set_char('A');
```

This code will set the discriminator value to '1', which selects the first field (field # 0) – the 'char' field. Then we access the char field, and set its value to 'A'.

18.3.3 Accessing Dynamic Type Data

CoreDX DDS provides a number of functions to access the data and examine the data type of a dynamic type instance. These functions are useful for subscribing applications. These applications read data from a discovered Topic where the data type was not known at compile time.

The following function in Table 18-2 returns the real type of the DynamicType.

Table 18-2: Function to Access the Real Type

Dynamic Type Function	Arguments	Return	Description
DynamicType_get_type()	(none)	DDS_TypeCodeKind	Returns the real type of this DynamicType.

The following functions in Table 18-3 return the value of a DynamicType instance. Possible values include all basic types listed in Table 11-3 in the Application Data Types Chapter.

Table 18-3: Functions to Access Values of Basic Types

Dynamic Type Function	Arguments	Return	Description
DynamicType_get_octet()	(none)	unsigned char	
DynamicType_get_char()	(none)	char	
DynamicType_get_short()	(none)	16-bit integer	
DynamicType_get_ushort()	(none)	16 bit unsigned int	
DynamicType_get_long()	(none)	32-bit integer	
DynamicType_get_ulong()	(none)	32-bit unsigned int	
DynamicType_get_longlong()	(none)	64-bit integer	
DynamicType_get_ulonglong()	(none)	64-bit unsigned int	
DynamicType_get_float()	(none)	float	
DynamicType_get_double()	(none)	double	
DynamicType_get_longdouble()	(none)	double	
DynamicType_get_string()	(none)	const char * (or String, for Java)	

The following functions in Table 18-4 help determine the structure of the constructed type.

Table 18-4: Functions to Access Constructed Types

Dynamic Type Function	Arguments	Return	Description
get_type()		DDS_TypeCodeKind	Return the kind of type for this DynamicType instance. For example, long.

get_max_length()		32-bit unsigned int	For arrays and fixed-length sequences and strings, return the maximum length.
get_length()		32-bit unsigned int	For arrays and sequences, return the length.
get_element_type()		DDS_DynamicType	For arrays and sequences, return the type of the elements.
get_element()	unsigned integer	DDS_DynamicType	For arrays and sequences, return a particular element.
get_num_fields()		32-bit unsigned int	For structures and unions, return the number of fields.
get_field()	integer	DDS_DynamicType	For structures and unions, return a particular field.
get_field_name()	integer	const char * (or String, for Java)	For structures and unions, return a particular field name.
get_discriminator()		DDS_DynamicType	For a union, return the discriminator.

18.4 Subscribe with Dynamic Types

A CoreDX DDS application may subscribe to a Topic that is discovered at run time, without any knowledge of the data type associated with the Topic. This is considered the manual use of Dynamic Types, since there is no dependence on prior knowledge of the data type. The basic steps involved in this type of application are:

1. Use the built-in DataReader to Discover a DataWriter
2. Use the Type Code information from the DataWriter to register a Data Type
3. Use the topic information from the DataWriter to create a Topic
4. Create a Dynamic DataReader
5. Read data

Table 18-5 contains the necessary source code for creating a Dynamic DataReader. The full code for this example can be found in the examples directory of the CoreDX DDS release.

Table 18-5: Example 'Manual' Dynamic Type DataReader

Example code for a Dynamic Type Data Reader

```
/* Create a DomainParticipant and Subscriber in the normal way */

/* Get access to the built-in "publication" data reader
 * so we can learn about discovered writers
 */

bi_subscriber = DDS_DomainParticipant_get_builtin_subscriber(domain);
bi_pub_reader = DDS_Subscriber_lookup_datareader(bi_subscriber,
                                                 "DCPSPublication");

/* Wait to discover a DataWriter */

while(!found)
  {
    DDS_DCPSPublicationPtrSeq  writers;
    DDS_SampleInfoSeq          samples_info;
    DDS_ReturnCode_t           retval;

    /* in case we don't see a writer this time around */

    toc_sleep(USEC_PER_SEC);

    /* Read from the Publications built-in topic, this is
     * where we can learn about discovered DataWriters
     */

    INIT_SEQ(writers);
    INIT_SEQ(samples_info);
    retval = DDS_DCPSPublicationDataReader_read ( bi_pub_reader,
                                                  &writers,
                                                  &samples_info,
                                                  DDS_LENGTH_UNLIMITED,
                                                  DDS_ANY_SAMPLE_STATE,
                                                  DDS_NEW_VIEW_STATE,
                                                  DDS_ALIVE_INSTANCE_STATE);
    if (retval == DDS_RETCODE_OK)
    {
        /* We might have found more than one, but just
```

```
            * use the first one we found.
            */

           DDS_DCPSPublication * pub  = writers._buffer[0];
           DDS_SampleInfo      * si   = samples_info._buffer[0];

           DDS_TypeDefinition       td  = NULL;
           DDS_TypeSupport          dts = NULL;

           /* Make sure we read about a writer (could be a
            * dispose or unregister notification with no
            * DataWriter information)
            */
           if ( si->valid_data)
             {
                /* make sure the writer published its 'TypeCode' */

                if (pub->typecode.value._length>0)
                  {
                    printf("Found a writer, creating TypeSupport.\n");
                    found    = 1;

                    /* Now, we construct a DynamicTypeTypeSupport
                     * to handle the discovered data type */
                    td       = DDS_create_type_definition(
                                        pub->typecode.value._buffer,
                                        pub->typecode.value._length,
                                        pub->typecode.encoding);
                    dts      = DDS_create_dynamic_typesupport( td );

                    /* Register the new  TypeSupport */
                    DDS_DomainParticipant_register_type(domain, dts,
                                                        pub->type_name);
                    /* Create a Topic associated based on that type */
                    topic = DDS_DomainParticipant_create_topic(domain,
                                       pub->topic_name, pub->type_name,
                                       DDS_TOPIC_QOS_DEFAULT, NULL, 0 );
                    /* create a DynamicTypeDataReader */
                    dr = DDS_Subscriber_create_datareader( subscriber,
                                        DDS_Topic_TopicDescription(topic),
                                        DDS_DATAREADER_QOS_DEFAULT,
                                        &drListener,
                                        DDS_DATA_AVAILABLE_STATUS );

                } /* end if the DataWriter had TypeCode information */

           } /* end if valid data */

           DDS_DCPSPublicationDataReader_return_loan( bi_pub_reader,
                                                &writers, &samples_info);
```

```
    } /* end if read() returned OK */

  } /* end while() we haven't discovered a writer */


/* Sleep for as long as necessary to receive data on drListener */

/* Cleanup in the normal way */
```

18.5 Publish with Dynamic Types

A CoreDX DDS application may publish to a Topic that is discovered at run time, without any knowledge of the data type associated with the Topic. This is considered the manual use of Dynamic Types, since there is no dependence on prior knowledge of the data type. The basic steps involved in this type of application are:

1. Create a dynamic data type representing the type of data to be published
2. Register the dynamic data type
3. Create a Topic
4. Create a Dynamic DataWriter
5. Initialize and send data

Table 18-5 contains the necessary source code for creating a Dynamic DataWriter. The full code for this example can be found in the examples directory of the CoreDX DDS release.

Example code for a Dynamic Type Data Writer

```
/* Create a DomainParticipant and Subscriber in the normal way */

/* Build a DynamicType structure representing our data type:
 *     string msg, long x
 */

simplemsg_dt = DDS_DynamicType_alloc_struct();
DDS_DynamicType_set_num_fields(simplemsg_dt, 2);
{
  DDS_DynamicType          temp_dt;

  /* data->msg: */
  temp_dt = DDS_DynamicType_alloc_string();
```

```
  DDS_DynamicType_set_field(simplemsg_dt, 0, "msg", temp_dt, 0);
  /* data->x: */
  temp_dt = DDS_DynamicType_alloc_basic(DDS_TYPECODE_LONG);
  DDS_DynamicType_set_field(simplemsg_dt, 1, "x", temp_dt, 0);
}

/* Construct a 'TypeDefinition' from the DynamicType */
td       = DDS_create_type_definition_from_dynamictype(simplemsg_dt);

DDS_DynamicType_free(simplemsg_dt); /* no longer needed */

/* Construct a 'DynamicTypeTypeSupport' for this 'simplemsg' type */
dts      = DDS_create_dynamic_typesupport( td );

/* Register the data type with the CoreDX middleware.
 * This is required before creating a Topic with
 * this data type.
 */
DDS_DomainParticipant_register_type(domain, dts, "SimpleMsg");

/* create a DDS Topic with the SimpleMsg data type. */
topic =
  DDS_DomainParticipant_create_topic(domain,
                                     "SimpleMsg",
                                     "SimpleMsg",
                                     DDS_TOPIC_QOS_DEFAULT,
                                     NULL,
                                     0 );

/* Create a DDS DataWriter on the simplemsg topic,
 * with default QoS settings and no listeners.
 */
dw = DDS_Publisher_create_datawriter(publisher,
                                       topic,
                                       DDS_DATAWRITER_QOS_DEFAULT,
                                       NULL,
                                       0 );

/*********************************************/
/*   construct sample and write it           */
/*********************************************/
simplemsg_data = DDS_DynamicType_alloc_struct();
DDS_DynamicType_set_num_fields(simplemsg_data, 2);
{
   DDS_DynamicType dt1;
   /* data->msg: */
   dt1 = DDS_DynamicType_alloc_string();
   DDS_DynamicType_set_string(dt1, "Hello WORLD from C!");
   DDS_DynamicType_set_field(simplemsg_data, 0, "msg", dt1, 0);
   /* data->x: */
```

```
    dt1 = DDS_DynamicType_alloc_basic(DDS_TYPECODE_LONG);
    DDS_DynamicType_set_long(dt1, 100);
    DDS_DynamicType_set_field(simplemsg_data, 1, "x", dt1, 0);
}

/* Write data in the normal way */

/* Cleanup in the normal way */
```

18.6 Dynamic Types and Generated Code

CoreDX DDS provides additional flexibility to application developers with Dynamic Types, allowing the mixed use of Dynamic Types and type specific code generated by the CoreDX DDL Compiler. This can be an attractive option when the data types are known at compile time, but it is desirable to keep the generated code to a minimum.

The CoreDX DDL Compiler can generate a small subset of type specific code that includes only specialized helper functions for working with Dynamic Types. This smaller set of generated code makes using Dynamic Types much easier.

The CoreDX baseline contains example code for using the Dynamic Types generated helper functions.

18.6.1 CoreDX DDL Compiler Special Options

The Dynamic Types helper functions are generated by the CoreDX DDL Compiler. In addition to any other options to the CoreDX DDL Compiler, use the following option:

```
-i t
```

This option causes the CoreDX DDL Compiler to generate only the Dynamic Type helper functions and related structures. This results in a much smaller set of generated code.

18.7 Compiling an Application using Dynamic Types

The support for Dynamic Types is located in a separate library. This allows us to keep the basic CoreDX DDS library extremely small. To compile an application that uses dynamic types, add **libdds_dyntype** to the library line in your Makefile, in addition to libdds or libdds_cf. The C++ and Java

libraries are still required if your application is using one of these language bindings. For example, on a Linux machine you could use the following:

```
LIBS= -Wl,-Bstatic -ldds_dyntype -ldds -Wl,-Bdynamic
```

Chapter 19 Threading Options

19.1 Overview

CoreDX DDS contains advanced multi-threaded technology. This feature allows any application (even non-threaded applications) running on multi-core hardware to make use of multiple cores. Using multiple cores on multi-core hardware provides significant performance benefits, as confirmed using Intel's Thread Checking benchmarking system.

Because many of our users are not running on multi-core hardware, and in fact are running on significantly reduced-power single core hardware, CoreDX DDS is configurable to provide performance benefits in this type of resource constrained environments.

19.2 Configuring Threading Options

CoreDX DDS runs in an optimized threaded model by default. This mode creates three threads for each DomainParticipant created in the application:

1. The main (application) thread

   The main (application) thread contains the main() of the application, and most of the application execution.

2. The CoreDX DDS reading thread

   The CoreDX DDS reading thread is responsible for reading data off the transport (UDP socket, or other transport as configured by the application). Data is read off the transport, unmarshaled, and put into the Data Caches of the appropriate DataReaders.

3. The CoreDX DDS work thread

   The CoreDX DDS work thread performs all remaining DDS "work". This includes discovery, writing application data to the transport, maintaining liveliness, performing handshaking and any necessary repairs for Reliable readers and writers, and application notification of events.

CoreDX DDS threading is configurable through the DomainParticipant's CoreDX_ThreadModelQosPolicy.

19.2.1 Single Threaded Configuration

CoreDX DDS features a *single threaded* model for higher performance on significantly resource constrained single threaded devices. Eliminating threads allows CoreDX DDS to eliminate much of the locking code, and reduce the amount of context switches required by the application, helping to reduce the required memory and CPU resources.

The single threaded model requires the application to periodically "hand over" CPU time to CoreDX DDS to perform it work (discovery, reading data, writing data, maintaining liveliness, etc.). Otherwise, this single threaded model uses the same API as the multi-threaded model. There are no new libraries to link with. This ensures there is not a completely new API to learn, and makes it easy to move applications from multi-threaded to single-threaded modes (and back again).

The CoreDX DDS release packages contain example code illustrating the use of the single threaded mode. This example can be found in the examples directory, "hello_nothr". If you look at hello_pub.c in this example, you will find that the 'no threads' programming model has a very simple API.

First, use the CoreDX_ThreadModelQoSPolicy on the DomainParticipant to configure CoreDX DDS to use the single threaded model.

QoS Policy	Default Value	Description
CoreDX_ThreadModelQosPolicy		
use_threads (unsigned char)	1 (true)	Setting use_threads to 0 (false) will configure CoreDX DDS to use the single threaded model.

Next, provide CoreDX DDS with time to perform work. It is important to provide CoreDX DDS with enough opportunities to run so that it can manage its internal tasks. This entails inserting calls to DDS_DomainParticipant_do_work() at strategic points in your 'main' program.

```
DomainParticipant::do_work( duration_t time )
```

The application provides CoreDX DDS a duration in which it can perform its internal work. The do_work call will return when this time expires.

19.2.2 Listener Thread Configuration

CoreDX DDS can create a 4th thread (only applicable when using the multi threaded model) that will handle application listener callbacks.

In the standard 3-thread threaded model, application listener callbacks are handled by the work thread (along with discovery, writing data, and maintaining liveliness). This means that a long-running application defined listener callback (for example, in a DataReader's on_data_available listener callback) can block CoreDX DDS from performing other internal tasks.

The solution for those applications that cannot reduce their listener callback functions is to create a separate thread for listener callbacks. With the listener thread enabled, an application can block inside a listener callback without effecting other DDS operations.

QoS Policy	Default Value	Description
CoreDX_ThreadModelQosPolicy		
create_listener_thread (unsigned char)	0 (false)	Setting create_listener_thread to 1 (true) will configure the DomainParticipant to create the 4th thread for application callbacks. This option is applicable only when use_threads is set to non-zero (true).

Chapter 20 Transmit Buffers

20.1 Overview

Each CoreDX DDS DataWriter contains a *transmit buffer* to hold data that is waiting to be published.

Transmit buffers usually hold sample data from DataWriter::write() calls, but can also hold instance lifecycle information including unregister or dispose actions. Both built-in DataWriters and application defined DataWriters contain these transmit buffers.

By default, transmit buffers are dynamic, that is, they grow and shrink as necessary to minimize the amount of memory consumed by the CoreDX DDS infrastructure. CoreDX DDS transmit buffers can be configured to be a static size, or configured to be dynamic with specified minimum and maximum sizes.

20.2 Dynamic Transmit Buffers

CoreDX DDS transmit buffers are by default, dynamic. Without any configuration, DataWriter transmit buffers will grow and shrink as necessary to support the size of data written while consuming a minimal amount of memory.

CoreDX DDS transmit buffer sizes can be configured with a QoS policy, or with environment variables. The DataWriter QoS policy to configure the minimum and maximum buffer sizes is described in the following table.

Table 20-1: Instance Example

QoS Policy	Default Value	Description
DataWriter_RTPSWriterQoSPolicy (application-definied DataWriters)		
min_buffer_size (unsigned int)	16	In bytes, the transmit buffer will start at this size, and in dynamic operations, will not shrink smaller

		than this size.
max_buffer_size (unsigned int)	65400	In bytes, the transmit buffer will not grow larger than this size.
DataWriter_RTPSWriterQoSPolicy (built-in DataWriters)		
min_buffer_size (unsigned int)	16	For built-in writers, the transmit buffer will start at this size, and in dynamic operations, will not shrink smaller than this size (in bytes).
max_buffer_size (unsigned int)	32768	For built-in writers, the transmit buffer will not grow larger than this size (in bytes).

The CoreDX DDS environment variables to configure the transmit buffer size are: COREDX_MIN_TX_BUFFER_SIZE and COREDX_MAX_TX_BUFFER_SIZE, and are used in the same manner as the DataWriter QoS policy described above. These environment variables will override the default sizes of all DDS entities (both built-in and application defined).

These are the sizes that bound the dynamic sizing of the buffers. The transmit buffer will not grow beyond max_buffer_size, and the transmit buffer will not shrink below min_buffer_size.

The maximum transmit buffer size will affect how CoreDX DDS aggregates, batches, and/or fragments written data. This is particularly noticeable with applications that perform many, frequent writes, or have bursts of writes. With a small upper bound on the transmit buffer, CoreDX DDS will need to perform many individual writes, rather than aggregating or batching samples together to be sent at one time.

When the application writes a sample that is larger than the configured maximum transmit buffer size for the DataWriter, CoreDX DDS will fragment the data sample as necessary to fit the transmit buffer, and re-assemble the sample on the receiving side before it is available to the receiving application.

The environment variable: COREDX_MAX_PACKET_SIZE (available in earlier CoreDX DDS releases) was equivalent to COREDX_MAX_TX_BUFFER_SIZE. The MAX_PACKET_SIZE environment variable is deprecated, and should no longer be used.

20.3 Static Transmit Buffers

Since allocating and de-allocating memory can be expensive operations, applications interested in very low latencies may benefit from a static transmit buffer that does not grow or shrink through the life of the application. This configuration is possible by setting the min_buffer_size and max_buffer_size to the same value, using either the QoS policies or environment variables described above.

Special care should be taken before setting a static transmit buffer size. Since the transmit buffer must be large enough to hold the complete marshaled data sample, it is important to understand the possible sizes for all possible application data samples written by the application. This is especially true when globally configuring static built-in DataWriter transmit buffers using the environment variables.

Chapter 21 Receive Buffers

21.1 Overview

Each CoreDX DDS DomainParticipant contains a *receive buffer* to hold incoming data that will be passed to its DataReaders and eventually the reading application.

Receive buffers are used to hold data read from the transport until is it processed (parsed) by the CoreDX DDS infrastructure. There is one receive buffer for each DomainParticipant (as opposed to one for each DataReader).

Receive buffers are dynamic, that is, they grow and shrink as necessary to minimize the amount of memory consumed by the CoreDX DDS infrastructure. The application may set the maximum size the DomainParticipant's receive buffer, limiting its growth. This is accomplished using an environment variable: COREDX_RX_BUFFER_SIZE. This environment variable should be set to the maximum buffer size in bytes. There is no limit imposed by CoreDX DDS on the maximum receive buffer size.

Chapter 22 Data Batching

Data Batching is the processes of combining data samples into one RTPS message in order to reduce the network overhead and improve throughput, especially with smaller samples.

By default, data batching is disabled in CoreDX DDS DataWriters. DataReaders are configured to accept batch messages by default. Applications can use the following QoS policies to configure *data batching*.

QoS Policy	Default Value	Description
CoreDX_RTPSWriterQosPolicy		
enable_batch_msg (unsigned char or boolean)	0 (or false)	Configure the DataWriter to use batching. Possible values are: 0 or 1 (false or true)
CoreDX_RTPSReaderQosPolicy		
accept_batch_msg (unsigned char or boolean)	1 (or true)	Configure the DataReader to use batching. Possible values are: 0 or 1 (false or true)

Chapter 23 Licensing

CoreDX DDS uses development and run-time licenses. A development license is required for using the CoreDX DDS DDL compiler (coredx_ddl). A run-time license is required for running an application built with the CoreDX DDS library. Both run-time and development licenses can be contained in the same license file or in separate files. Here is an example license file containing both development and run-time licenses:

coredx.lic

```
#=====================================================================
# CoreDX DDS License file for CompanyX
#
# Created:  Jul 22, 2008, by Twin Oaks Computing, Inc.
# Contains: 2 development licenses, 2 run-time licenses
#
#=====================================================================
#
# development LICENSE lines:
#     - Contain your development keys - DO NOT EDIT!
LICENSE PRODUCT=coredx_ddl BUILD=Release OS=linux ARCH=x86 USERID=ntucker
HOSTID=00195b70c3be CUSTOMER=Company_X SIG=abcdefghijklmnopqrstuvwxyz
#
# run-time LICENSE lines:
#
#     - Contain your run-time keys - DO NOT EDIT!
LICENSE PRODUCT=coredx_c BUILD=Release HOSTID=00195b70c3be
CUSTOMER=Company_X SIG= abcdefghijklmnopqrstuvwxyz
```

Figure 23-1: Example CoreDX DDS license file

23.1 Development Licenses

Development licenses are contained in a license file. To develop (compile) with CoreDX DDS, an environment variable `TWINOAKS_LICENSE_FILE` must be set to the full path to the license file.

23.2 Run-time Licenses

There are a few ways to use run-time licenses. Run-time licenses may be contained in a license file, or otherwise coded into the application and provided to CoreDX DDS through the license API.

1. Use an Environment Variable

   The environment variable `TWINOAKS_LICENSE_FILE` may be set to one of the following:

   - The full path to the license file
   - The LICENSE string containing the run-time license

   If you have access to the license file from the run-time environment, the simplest way to use the license is to set a TWINOAKS_LICENSE_FILE environment variable to be the full path to the license file.

   If you do not have access to the license file, you can still use the license by setting the TWINOAKS_LICENSE_FILE environment variable to the appropriate run-time LICENSE line. The run-time license line starts with the following:

   ```
   LICENSE PRODUCT=coredx_c
   ```

   Using the license string is a good option for embedded run-time environments. For the run-time license in the above example license file, set your TWINOAKS_LICENSE_FILE environment variable like:

   ```
   linux% export TWINOAKS_LICENSE_FILE=<LICENSE
   PRODUCT=coredx_c HOSTID=00195b70c3be
   CUSTOMER=Company_X SIG=abcdefghijklmnopqrstuvwxyz>
   ```

2. Use the API

   The DomainParticipantFactory provides an API to set the license string:

```
DomainParticipantFactory::set_license(const char * lic)
```

   The `lic` argument may be set to any of the options that can be used for the TWINOAKS_LICENSE_FILE environment variable, described above. That is one of the following:

- The full path to the license file
- The LICENSE string containing the run-time license

The license API is particularly useful for operating systems that do not support environment variables. This allows the application to obtain the license string in any manner acceptable by the environment and system

requirements, and then use the API to pass the license string to CoreDX DDS.

Chapter 24 Troubleshooting

24.1 General Troubleshooting Tools

Network communication can be complex to troubleshoot. It is recommended that the developer become familiar with standard tools available on the development network. For example, under UNIX, tools such as **ifconfig**, **netstat**, and **route** can be useful to gain an understanding of the network configuration. Further, tools that capture and decode network traffic are very useful. The **wireshark** tool has wide platform support and includes a protocol analyzer for RTPS (the DDS wire protocol). **Wireshark** is an indispensible tool for analyzing DDS network traffic. (See www.wireshark.org).

Twin Oaks Computing offers a tool that is specially designed for analyzing and debugging DDS applications: CoreDX DDS Spy. The CoreDX DDS Spy tool displays, at a glance, all the DDS Entities on the network. This allows the application developer to quickly view of all the DataReaders and DataWriters on the network, what Topic they are communicating on, and which ones are not communicating due to QoS or data type mismatches. In addition, CoreDX DDS Spy to view all the DDS network traffic, including samples written by DDS application for further analysis of DDS applications.

24.2 No Communications between DDS applications

If Readers and Writers are not communicating at all, then there are several items to check. First, it is recommended that Listeners be installed on both the reader and writer to handle all of the events. These events may provide insight into why the entities are not communicating. For example, the requested_incompatible_qos and offered_incompatible_qos listeners are very useful.

24.2.1 Network Configuration (if running across a network)

If all your DDS DomainParticipants are running on one machine, skip this section.

If the DDS DomainParticipants are running across a network, is your network working? Can you use *ping* or another program to talk between your hosts?

24.2.2 Discovery

The first step in DDS communications is the *discovery* process, where DomainParticipants broadcast their existence and look for peer DomainParticipants. This discovery protocol uses *multicast*.

If your DDS DomainParticipants are communication across routers or a Virtual Machine, you may need to increase the reach of your multicast packets by increasing the number of hops (see the *CoreDX DDS Transport* Chapter for more information).

24.2.3 DataReader / DataWriter matching

The next step in DDS communications is matching a DataWriter to a DataReader. Matching requires several compatible attributes:

1. The Topic name must match. Carefully check the create_topic, create_datawriter, and create_datareader calls to ensure the same Topic name string is used for both the DataWriter's Topic and the DataReader's Topic.
2. The data types must match. Not only the name of the data type, but also the *types* must match. CoreDX DDS serializes the type of the data into a "typecode", and compares the typecode of the DataWriter with the typecode of the DataReader. These types must match.
3. Recall that the QoS setting for the DataReader and DataWriter must be compatible for communications to occur (see the *QoS Compatibility* section).

The SubscriptionMatchedStatus and PublicationMatchedStatus statuses record matching DataReaders and DataWriters. The OfferedIncompatibleQosStatus and RequestedIncompatibleQosStatus record mis-matching DataWriters and DataReaders due to QoS incompatibility. Use Listeners (see the *Listeners* section) or Conditions (see the *Conditions and WaitSets* section) to check these statuses.

24.3 Missing or lost samples

There are numerous QoS policies that can cause samples to be missing or lost. A few of the more common ones are described below. In addition, QoS settings can interact with each other causing non-intuitive application behavior. While the examples below describe some common problems and solutions, your specific network environment and other QoS settings may result in application behavior different than what is described below.

Twin Oaks Computing is dedicated to the helping customers get the most out of their application communications using CoreDX DDS. Please contact us for additional support with your specific application.

24.3.1 Reliability

If you are communicating over a network, a slow or unreliable network can cause packets to be lost. Similarly, one "slow" subscribing host can have trouble keeping up with publishing hosts. Setting the Reliability QoS policy to RELIABLE can reduce or eliminate lost packets in this scenario.

It is important to note that a RELIABLE Reliability can only happen while both the DataWriter and DataReader are both in existence. Sometimes, a publishing application will exit (killing the DataWriter) before the DataReader has received all the published samples, resulting in lost samples.

24.3.2 Durability

If your DataReader is consistently missing the first one or two samples published by a DataWriter, chances are the discovery process is not completing (matching the DataWriter to the DataReader) before those first samples are written. In general, the solution is to raise the Durability QoS setting to TRANSIENT_LOCAL. This can have other side effects when combined with other QoS settings (see the *HISTORY* section). Other options include having the publishing application wait for a discovered DataReader or simply pause for one or two seconds before starting to write data; allowing the discovery process to complete.

24.3.3 History

By default, the History QoS policy is set to KEEP_LAST, with a depth of 1 (one). Consider a DataWriter writing samples fast enough that the CoreDX DDS infrastructure must queue any before sending, or a DataReader receiving samples fast enough that they must be queued before a read() or take() operation is used. With a History that is only keeping the 1 most recent data sample for each instance, there is a possibility for samples to be dropped. The solution is to increase the History depth to greater than 1, or set the History to KEEP_ALL.

There is only one combination of QoS settings that will guarantee samples are not lost during operations and that is:

Reliability = RELIABLE

History = KEEP_ALL

Resource Limits = Set

This combination of QoS policies forces the publishing application to block on a DataWriter::write() operation, if any matched DataReaders is unable to accept another sample. The DataWriter::write operation will complete (and return) once all matched DataReaders have enough room to receive an additional sample.

24.3.4 Putting it all together: Guaranteed Delivery

A common question from DDS users is, "How do I guarantee delivery with DDS?" The goal is to guarantee all data written by a publishing application will be delivered to all matched subscribing applications.

There are three QoS policies that need to be configured to guarantee delivery of data samples:

1. Reliability (kind = RELIABLE)
2. History (kind = KEEP_ALL)
3. Resource Limits (set to something other than infinite)

All of these QoS policies must be set to ensure delivery of published data.

The RELIABLE Reliability setting allows CoreDX DDS to monitor data reception, and retransmit data if it is not received by any DataReaders.

The KEEP_ALL History setting instructs CoreDX DDS that it is NOT OK to overwrite any data samples in the DataWriter's cache. This is important if there are any DataReaders that are having trouble "keeping up", and lots of samples must be stored for retransmission.

Setting the Resource Limits allows CoreDX DDS to limit the growth of the DataWriter's cache, even with the KEEP_ALL History setting. This is important, not only for resource utilization at run time, but also because it allows the application's call to DataWriter::write() to block if there is no more room in the cache (because at least one DataWriter has not acknowledged a number of sent samples).

24.4 TypeSupport version mismatch

CoreDX DDS provides strong data typing. Application developers define the data types that will be used for DDS communications at compile time, and use the CoreDX DDL compiler to generate type specific DDS code for each data type. This generated code interacts very closely with the DDS library to perform type specific operations (for example, serializing data on a write() and de-serializing data on a read()). For this reason, it is important that the DDL compiler used to generate the code match the DDS library that is linked into the application. If these versions do not match, CoreDX DDS will print a warning message when register_type() is called:

Sample Warning Message for Version Mismatch

```
WARNING:  MyType TypeSupport version does not match CoreDX
Library version.
This may cause software instability or crashes.
```

Figure 24-1: Example CoreDX DDS license file

To resolve this issue, re-generate your type specific code with the correct version of the CoreDX DDL compiler, and check the version of libdds.a that you are linking with your application.

24.5 Can't find it here?

Call us at 720-733-7906, send an email to support@twinoakscomputing.com, or check out our Frequently Asked Questions at http://www.twinoakscomputing.com/coredx/faq, or visit our online forums at http://www.twinoakscomputing.com/forums.

Chapter 25 About Twin Oaks Computing

Twin Oaks Computing, Inc is a company dedicated to developing and delivering quality software solutions. We leverage our technical experience and abilities to provide innovative and useful services in the domain of intelligence systems.

Twin Oaks Computing specializes in high-performance and embedded communications solutions for commercial and DoD applications. Our CoreDX DDS was first released in 2008. In March 2009, Twin Oaks Computing participated in the first public multi-vendor DDS interoperability demonstration. For more information on our products, please visit our website at http://www.twinoakscomputing.com.

Twin Oaks Computing is headquartered in Castle Rock, CO. Our staff has over 30 years of experience developing and supporting DoD systems. We have performed installs and upgrades of critical mission systems at U.S. military facilities around the world. Through this experience, we understand the importance of the systems that collect, manage, and distribute information for the warfighter.

We apply our technical experience to develop solutions in the following Intelligence Domains:

- Tactical Communications - Link 16, IBS, Link 11, Link 11B
- Tactical Data Correlation - Single and Multi-INT Correlation
- Situational Awareness - consolidated display of tactical data

We have Technical experience in the following areas:

- Networking - Ethernet, IP, UDP, TCP, RDMA
- Device Drivers - MILSTD-1553, Serial, Network Interface
- Interprocess Communication - DDS, Sockets, CORBA, RPC, SysV IPC
- Operating Systems - Solaris™, Linux™, FreeBSD™, VxWorks™, and others
- Database Technologies - Sybase™, Oracle™, MySQL™, and others

- Network Services - email servers, HTTP servers, DNS servers, firewalls
- System Security - DCID 6/3 security accreditation
- System Administration - scripting languages, backup/restore, storage management, software installation/configuration

We would be happy to discuss how we can help you. Please contact us at contact@twinoakscomputing.com.

Chapter 26 Contact Information

Have a question? Don't hesitate to contact us by any means convenient for you:

Web Site: http://www.twinoakscomputing.com

Email: support@twinoakscomputing.com

Twitter: @CoreDX_DDS

Phone: 720.733.7906

+33(0)9 62 23 72 20

Address:

755 Maleta Lane

Suite 203

Castle Rock, CO. 80108