Eclipse Cyclone DDS

Release 0.1.0

Eclipse Cyclone DDS project

Feb 14, 2019

Contents

CHAPTER 1

Installing Eclipse Cyclone DDS

1.1 System requirements

At the time of writing, Eclipse Cyclone DDS is known to run on Linux, macOS and Windows. The build-process is not yet able to generate native packages.

1.2 Linux and macOS

1.2.1 Post install steps

The installation package installs examples in system directories. In order to have a better user experience when building the Eclipse Cyclone DDS examples, it is advised to copy the examples to a user-defined location. This is to be able to build the examples natively and experiment with the example source code.

For this, the installation package provides the vdds_install_examples script, located in /usr/bin.

Create an user writable directory where the examples should go. Navigate to that directory and execute the script. Answer 'yes' to the questions and the examples will be installed in the current location.

Type `vdds_install_examples -h` for more information.

Paths

To be able to run Eclipse Cyclone DDS executables, the required libraries (like libddsc.so) need to be available to the executables. Normally, these are installed in system default locations and it works out-of-the-box. However, if they are not installed in those locations, it is possible that the library search path has to be changed. This can be achieved by executing the command:

```
export LD_LIBRARY_PATH=<install_dir>/lib:$LD_LIBRARY_PATH
```

1.3 Windows

1.3.1 Paths

To be able to run Eclipse Cyclone DDS executables, the required libraries (like ddsc.dll) need to be available to the executables. Normally, these are installed in system default locations and it works out-of-the-box. However, if they are not installed on those locations, it is possible that the library search path has to be changed. This can be achieved by executing the command:

```
set PATH=<install_dir>/bin;%PATH%
```

1.4 Test your installation

Eclipse Cyclone DDS includes a simple *Hello World!* application which can be run in order to test your installation. The *Hello World!* application consists of two executables: a so called HelloworldPublisher and a HelloworldSubscriber.

To run the example application, please open two console windows and navigate to the appropriate directory in both console windows. Run the HelloworldSubscriber in one of the console windows by the typing following command:

Windows `HelloworldSubscriber.exe`

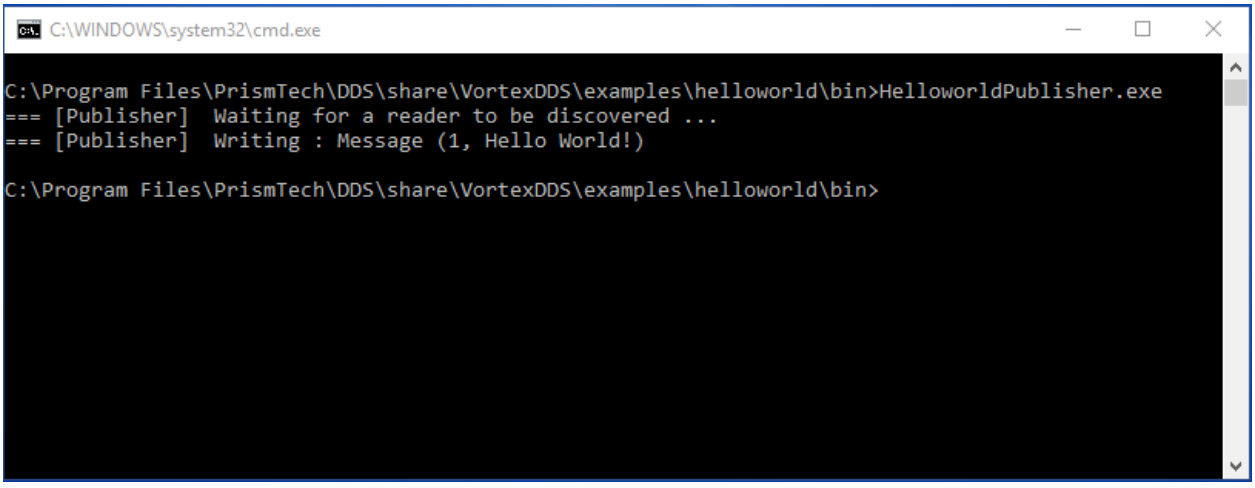
Linux `./HelloworldSubscriber`

and the HelloworldPublisher in the other console window by typing:

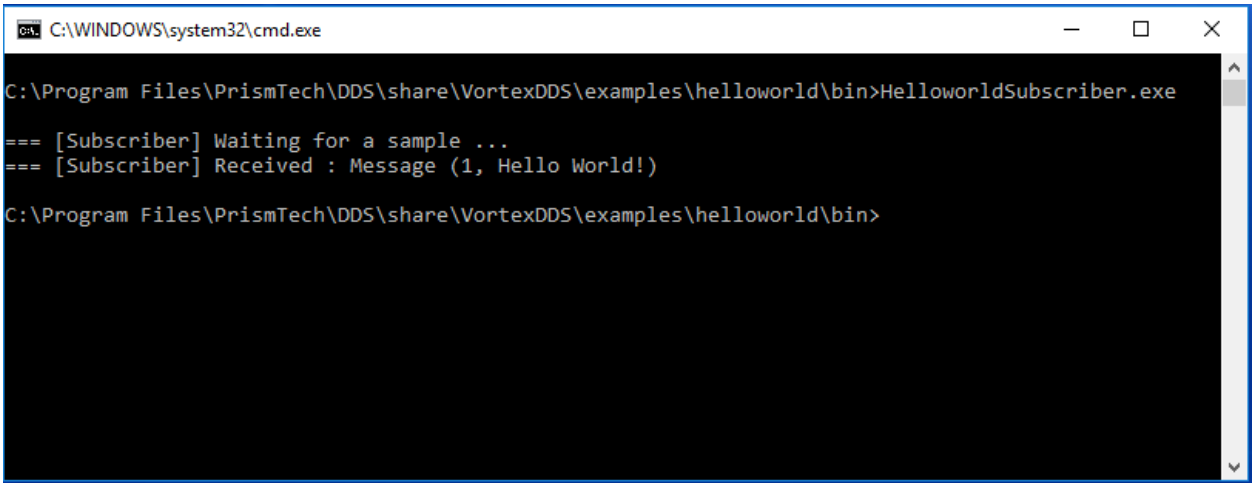
Windows `HelloworldPublisher.exe`

Linux `./HelloworldPublisher`

The output HelloworldPublisher should look like



while the HelloworldSubscriber will be looking like this



For more information on how to build this application your own and the code which has been used, please have a look at the *Hello World!* chapter.

CHAPTER 2

Building Eclipse Cyclone DDS applications

2.1 Building the *Hello World!* example

To test the *installation*, a small *Hello World!* application is used. This application will also be used as an introduction to DDS.

This chapter explains how to build this example, without details regarding the source code. The next chapter will explain what has to be done to code the *Hello World!* example.

The procedure used to build the *Hello World!* example can also be used for building your own applications.

Windows ...

Linux It is advised to have copied the Eclipse Cyclone DDS examples to a user-friendly location as described in *this* paragraph when actively building the Eclipse Cyclone DDS examples on Linux. This chapter refers to the Eclipse Cyclone DDS examples installed in the user-defined location.

2.1.1 Build Files

Three files are available *Hello World!* root directory to support building the example. Both *Windows native* (HelloWorld.sln) and *Linux native* (Makefile) build files will only be available for this *Hello World!* example. All the other examples make use of the *CMake* build system and thus only have the CMakeLists.txt build related file.

2.1.2 Linux Native Build

A Linux native `Makefile` is provided in the `examples/helloworld` directory within the destination location entered in the *vdds_install_examples script*. In a terminal, go to that directory and type

```
make
```

The build process should have access to the include files and the ddsc library. The Makefile expects them to be present at system default locations so that it can find them automatically. If this isn't the case on your machine, then please update the commented out `CFLAGS` and `LDFLAGS` within the `Makefile` to point to the proper locations.

This will build the HelloworldSubscriber and HelloworldPublisher executables in the helloworld source directory (not the bin directory that contains the pre-build binaries).

The *Hello World!* example can now be executed, like described in *Test your installation*, using the binaries that were just build. Be sure to use the right directories.

2.1.3 Windows Native Build

For the Windows Native Build, a Visual Studio solution file is available in the `examples/helloworld` directory. Use a file explorer to navigate to that directory and double click on the `HelloWorld.sln` file. Visual Studio should now start with the HelloWorld solution that contains three projects.

Project	Description
HelloWorldPublisher	Information to build the example publisher.
HelloWorldSubscriber	Information to build the example subcriber.
HelloWorldType	Information to (re)generate *HelloWorldData_Msg* data type.

Creating the *Hello World!* example executables is as simple as selecting the required configuration and building the solution.

`helloworld\vs\directories.props` contains the location of where the Eclipse Cyclone DDS header files and libraries are be placed. These locations are based on the default installation directory structure. When Eclipse Cyclone DDS is installed in a different directory, the following paths in `helloworld\vs\directories.props` should be changed, like:

```
<CycloneDDS_lib_dir>C:/Path/To/CycloneDDS/Installation/lib</CycloneDDS_lib_dir>
<CycloneDDS_inc_dir>C:/Path/To/CycloneDDS/Installation/include</CycloneDDS_inc_dir>
<CycloneDDS_idlc_dir>C:/Path/To/CycloneDDS/Installation/share/CycloneDDS/idlc</
↪CycloneDDS_idlc_dir>
```

To run the example, Visual Studio should run both the publisher and subscriber simultaneously. It is capable of doing so, but it's not its default setting. To change it, open the HelloWorld solution property page by right clicking the solution and selecting `Properties`. Then go to `Common Properties` -> `Startup Project`, select `Multiple startup project` and set `Action "Start"` for HelloWorldPublisher and HelloWorldSubscriber. Finish the change by selecting `OK`.

Visual Studio is now ready to actually run the *Hello World!* example, which can be done by selecting `Debug` -> `Start without debugging`. Both the HelloworldSubscriber and the HelloworldPublisher will be started and the HelloworldPublisher will write a message that is received by the HelloworldSubscriber.

2.2 Building With CMake

In the earlier chapters, building the *Hello World!* example is done natively. However, the *Hello World!* example can also be build using the CMake tool. This is what is recommended. In fact, all the other examples don't provide native makefiles, only CMake files.

2.2.1 CMake

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.

In other words, CMake's main strength is build portability. CMake uses the native tools, and other than requiring itself, does not require any additional tools to be installed. The same CMake input files will build with GNU make, Visual studio 6,7,8 IDEs, borland make, nmake, and XCode.

An other advantage of CMake is building out-of-source. It simply works out-of-the-box. There are two important reasons to choose this:

1. Easy cleanup (no cluttering the source tree). Simply remove the build directory if you want to start from scratch.
2. Multiple build targets. It's possible to have up-to-date Debug and Release targets, without having to recompile the entire tree. For systems that do cross-platform compilation, it is easy to have up-to-date builds for the host and target platform.

There are a few other benefits to CMake, but that is out of the scope of this document.

2.2.2 Hello World! CMake (CycloneDDS Package)

After the CMake digression, we're back with the *Hello World!* example. Apart from the native build files, CMake build files are provided as well. See `examples/helloworld/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.5)

if (NOT TARGET CycloneDDS::ddsc)
    # Find the CycloneDDS package. If it is not in a default location, try
    # finding it relative to the example where it most likely resides.
    find_package(CycloneDDS REQUIRED PATHS "${CMAKE_SOURCE_DIR}/../../")
endif()

# This is a convenience function, provided by the CycloneDDS package,
# that will supply a library target related the the given idl file.
# In short, it takes the idl file, generates the source files with
# the proper data types and compiles them into a library.
idlc_generate(HelloWorldData_lib "HelloWorldData.idl")

# Both executables have only one related source file.
add_executable(HelloworldPublisher publisher.c)
add_executable(HelloworldSubscriber subscriber.c)

# Both executables need to be linked to the idl data type library and
# the ddsc API library.
target_link_libraries(HelloworldPublisher HelloWorldData_lib CycloneDDS::ddsc)
target_link_libraries(HelloworldSubscriber HelloWorldData_lib CycloneDDS::ddsc)
```

It will try to find the `CycloneDDS` CMake package. When it has found it, every path and dependencies are automatically set. After that, an application can use it without fuss. CMake will look in the default locations for the code:*CycloneDDS* package.

The `CycloneDDS` package provides the `ddsc` library that contains the DDS API that the application needs. But apart from that, it also contains helper functionality (`idlc_generate`) to generate library targets from IDL files. These library targets can be easily used when compiling an application that depends on a data type described in an IDL file.

Two applications will be created, `HelloworldPublisher` and `HelloworldSubscriber`. Both consist only out of one source file.

Both applications need to be linked to the `ddsc` library in the `CycloneDDS` package and `HelloWorldData_lib` that was generated by the call to `idlc_generate`.

2.2.3 Hello World! Configuration

The *Hello World!* example is prepared to be built by CMake through the use of its `CMakeLists.txt` file. The first step is letting CMake configure the build environment.

It's good practice to build examples or applications out-of-source. In order to do that, create a `build` directory in the `examples/helloworld` directory and go there, making our location `examples/helloworld/build`.

Here, we can let CMake configure the build environment for us by typing:

```
cmake ../
```

Note: CMake does a pretty good job at guessing which generator to use, but some environments require that you supply a specific generator. For example, only 64-bit libraries are shipped for Windows, but CMake will generate a 32-bit project by default, resulting in linker errors. When generating a Visual Studio project keep in mind to append **Win64** to the generator. The example below shows how to generate a Visual Studio 2015 project.

```
cmake -G "Visual Studio 14 2015 Win64" ..
```

Note: CMake generators can also create IDE environments. For instance, the "Visual Studio 14 2015 Win64" will generate a Visual Studio solution file. Other IDE's are also possible, like Eclipse.

CMake will use the CMakeLists.txt in the helloworld directory to create makefiles that fit the native platform.

Since everything is prepared, we can actually build the applications (HelloworldPublisher and HelloworldSubscriber in this case).

2.2.4 Hello World! Build

After the configuration step, building the example is as easy as typing:

```
cmake --build .
```

Note: On Windows, it is likely that you have to supply the config of Visual Studio:

```
cmake --build . --config "Release"
```

while being in the build directory created during the configuration step: `examples/helloworld/build`.

The resulting Publisher and Subscriber applications can be found in:

Windows `examples\helloworld\build\Release`.

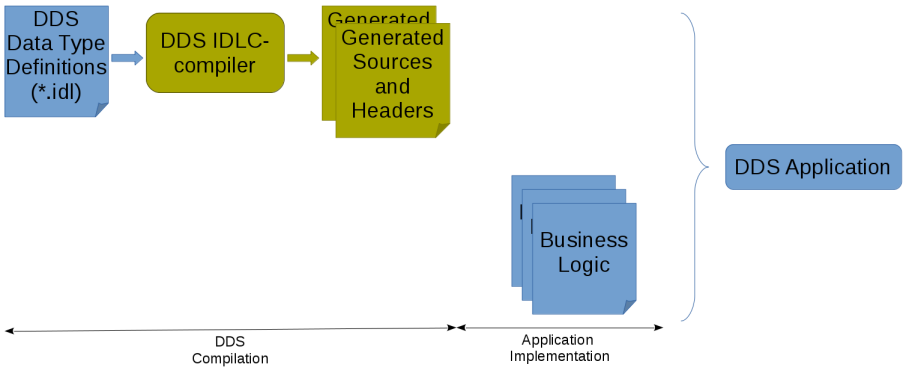
Linux `examples/helloworld/build`.

The *Hello World!* example can now be executed, like described in *Test your installation*, using the binaries that were just build. Be sure to use the right directories.

2.3 Summary

We've seen that a Eclipse Cyclone DDS application can be build by using a Makefile on Linux or a Visual Studio Solutions on Windows. Also CMake can be used to build a Eclipse Cyclone DDS application. In fact, it is the preferred way of building.

In the end, a predefined way of generating and building the source code should be followed when building Eclipse Cyclone DDS applications. The figure below shows how a typical Eclipse Cyclone DDS application is build.



Next chapter will provide an overview of all steps mentioned in the figure above.

CHAPTER 3

Hello World! in more detail

The previous chapter focused on building the *Hello World!* example while this chapter will focus on the code itself; what has to be done to code this small example.

3.1 Hello World! DataType

3.1.1 Data-Centric Architecture

By creating a Data-centric architecture, you get a loosely coupled information-driven system. It emphasizes a data layer that is common for all distributed applications within the system. Because there is no direct coupling among the applications in the DDS model, they can be added and removed easily in a modular and scalable manner. This makes that the complexity of a data-centric architecture doesn't really increase when more and more publishers/subscribers are added.

The *Hello World!* example has a very simple 'data layer' of only one data type `HelloWorldData_Msg` (please read on). The subscriber and publisher are not aware of each other. The former just waits until somebody provides the data it requires, while the latter just publishes the data without considering the number of interested parties. In other words, it doesn't matter for the publisher if there are none or multiple subscribers (try running the *Hello World!* example by starting multiple HelloworldSubscribers before starting a HelloworldPublisher). A publisher just writes the data. The DDS middleware takes care of delivering the data when needed.

3.2 HelloWorldData.idl

To be able to sent data from a writer to a reader, DDS needs to know the data type. For the *Hello World!* example, this data type is described using IDL and is located in HelloWorldData.idl. This IDL file will be compiled by a IDL compiler which in turn generates a C language source and header file. These generated source and header file will be used by the HelloworldSubscriber and HelloworldPublisher in order to communicate the *Hello World!* message between the HelloworldPublisher and the HelloworldSubscriber.

3.2.1 Hello World! IDL

There are a few ways to describe the structures that make up the data layer. The HelloWorld uses the IDL language to describe the data type in HelloWorldData.idl:

```
module HelloWorldData
{
  struct Msg
  {
    long userID;
    string message;
  };
  #pragma keylist Msg userID
};
```

An extensive explanation of IDL lies outside the scope of this example. Nevertheless, a quick overview of this example is given anyway.

First, there's the `module HelloWorldData`. This is a kind of namespace or scope or similar. Within that module, there's the `struct Msg`. This is the actual data structure that is used for the communication. In this case, it contains a `userID` and `message`.

The combination of this module and struct translates to the following when using the c language.

```
typedef struct HelloWorldData_Msg
{
  int32_t userID;
  char * message;
} HelloWorldData_Msg;
```

When it is translated to a different language, it will look different and more tailored towards that language. This is the advantage of using a data oriented language, like IDL, to describe the data layer. It can be translated into different languages after which the resulting applications can communicate without concerns about the (possible different) programming languages these application are written in.

3.2.2 Generate Sources and Headers

Like already mentioned in the *Hello World! IDL* chapter, an IDL file contains the description of data type(s). This needs to be translated into programming languages to be useful in the creation of DDS applications.

To be able to do that, there's a pre-compile step that actually compiles the IDL file into the desired programming language.

A java application `org.eclipse.cyclonedds.compilers.Idlc` is supplied to support this pre-compile step. This is available in `idlc-jar-with-dependencies.jar`

The compilation from IDL into c source code is as simple as starting that java application with an IDL file. In the case of the *Hello World!* example, that IDL file is HelloWorldData.idl.

```
java -classpath "<install_dir>/share/CycloneDDS/idlc/idlc-jar-with-dependencies.jar"␣
↪org.eclipse.cyclonedds.compilers.Idlc HelloWorldData.idl
```

Windows The `HelloWorldType` project within the HelloWorld solution.

Linux The `make datatype` command.

This will result in new `generated/HelloWorldData.c` and `generated/HelloWorldData.h` files that can be used in the *Hello World!* publisher and subscriber applications.

The application has to be rebuild when the data type source files were re-generated.

Again, this is all for the native builds. When using CMake, all this is done automatically.

3.2.3 HelloWorldData.c & HelloWorldData.h

As described in the *Hello World! DataType* paragraph, the IDL compiler will generate this source and header file. These files contain the data type of the messages that are sent and received.

While the c source has no interest for the application developers, HelloWorldData.h contains some information that they depend on. For example, it contains the actual message structure that is used when writing or reading data.

```
typedef struct HelloWorldData_Msg
{
    int32_t userID;
    char * message;
} HelloWorldData_Msg;
```

It also contains convenience macros to allocate and free memory space for the specific data types.

```
HelloWorldData_Msg__alloc()
HelloWorldData_Msg_free(d,o)
```

It contains an extern variable that describes the data type to the DDS middleware as well.

```
HelloWorldData_Msg_desc
```

3.3 Hello World! Business Logic

Apart from the *HelloWorldData data type files* that the *Hello World!* example uses to send messages, the *Hello World!* example also contains two (user) source files (*subscriber.c* and *publisher.c*), containing the business logic.

3.3.1 *Hello World!* Subscriber Source Code

Subscriber.c contains the source that will wait for a *Hello World!* message and reads it when it receives one.

```
#include "ddsc/dds.h"
#include "HelloWorldData.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* An array of one message (aka sample in dds terms) will be used. */
#define MAX_SAMPLES 1

int main (int argc, char ** argv)
{
    dds_entity_t participant;
    dds_entity_t topic;
    dds_entity_t reader;
    HelloWorldData_Msg *msg;
    void *samples[MAX_SAMPLES];
    dds_sample_info_t infos[MAX_SAMPLES];
```

(continues on next page)

(continued from previous page)

```
    dds_return_t ret;
    dds_qos_t *qos;
    (void)argc;
    (void)argv;

    /* Create a Participant. */
    participant = dds_create_participant (DDS_DOMAIN_DEFAULT, NULL, NULL);
    DDS_ERR_CHECK (participant, DDS_CHECK_REPORT | DDS_CHECK_EXIT);

    /* Create a Topic. */
    topic = dds_create_topic (participant, &HelloWorldData_Msg_desc,
                              "HelloWorldData_Msg", NULL, NULL);
    DDS_ERR_CHECK (topic, DDS_CHECK_REPORT | DDS_CHECK_EXIT);

    /* Create a reliable Reader. */
    qos = dds_create_qos ();
    dds_qset_reliability (qos, DDS_RELIABILITY_RELIABLE, DDS_SECS (10));
    reader = dds_create_reader (participant, topic, qos, NULL);
    DDS_ERR_CHECK (reader, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
    dds_delete_qos(qos);

    printf ("\n=== [Subscriber] Waiting for a sample ...\n");

    /* Initialize sample buffer, by pointing the void pointer within
     * the buffer array to a valid sample memory location. */
    samples[0] = HelloWorldData_Msg__alloc ();

    /* Poll until data has been read. */
    while (true)
    {
        /* Do the actual read.
         * The return value contains the number of read samples. */
        ret = dds_read (reader, samples, infos, MAX_SAMPLES, MAX_SAMPLES);
        DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);

        /* Check if we read some data and it is valid. */
        if ((ret > 0) && (infos[0].valid_data))
        {
            /* Print Message. */
            msg = (HelloWorldData_Msg*) samples[0];
            printf ("=== [Subscriber] Received : ");
            printf ("Message (%d, %s)\n", msg->userID, msg->message);
            break;
        }
        else
        {
            /* Polling sleep. */
            dds_sleepfor (DDS_MSECS (20));
        }
    }

    /* Free the data location. */
    HelloWorldData_Msg_free (samples[0], DDS_FREE_ALL);

    /* Deleting the participant will delete all its children recursively as well. */
    ret = dds_delete (participant);
    DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
```

(continues on next page)

(continued from previous page)

```
75
76      return EXIT_SUCCESS;
77  }
```

We will be using the DDS API and the *HelloWorldData_Msg* type to receive data. For that, we need to include the appropriate header files.

```
#include "ddsc/dds.h"
#include "HelloWorldData.h"
```

The main starts with defining a few variables that will be used for reading the *Hello World!* message. The entities are needed to create a reader.

```
dds_entity_t participant;
dds_entity_t topic;
dds_entity_t reader;
```

Then there are some buffers that are needed to actually read the data.

```
HelloWorldData_Msg *msg;
void *samples[MAX_SAMPLES];
dds_sample_info_t info[MAX_SAMPLES];
```

To be able to create a reader, we first need a participant. This participant is part of a specific communication domain. In the *Hello World!* example case, it is part of the default domain.

```
participant = dds_create_participant (DDS_DOMAIN_DEFAULT, NULL, NULL);
```

The another requisite is the topic which basically describes the data type that is used by the reader. When creating the topic, the *data description* for the DDS middleware that is present in the *HelloWorldData.h* is used. The topic also has a name. Topics with the same data type description, but with different names, are considered different topics. This means that readers/writers created with a topic named "A" will not interfere with readers/writers created with a topic named "B".

```
topic = dds_create_topic (participant, &HelloWorldData_Msg_desc,
                          "HelloWorldData_Msg", NULL, NULL);
```

When we have a participant and a topic, we then can create the reader. Since the order in which the *Hello World!* Publisher and *Hello World!* Subscriber are started shouldn't matter, we need to create a so called 'reliable' reader. Without going into details, the reader will be created like this

```
dds_qos_t *qos = dds_create_qos ();
dds_qset_reliability (qos, DDS_RELIABILITY_RELIABLE, DDS_SECS (10));
reader = dds_create_reader (participant, topic, qos, NULL);
dds_delete_qos(qos);
```

We are almost able to read data. However, the read expects an array of pointers to valid memory locations. This means the samples array needs initialization. In this example, we have an array of only one element: `#define MAX_SAMPLES 1`. So, we only need to initialize one element.

```
samples[0] = HelloWorldData_Msg__alloc ();
```

Now everything is ready for reading data. But we don't know if there is any data. To simplify things, we enter a polling loop that will exit when data has been read.

Within the polling loop, we do the actual read. We provide the initialized array of pointers (`samples`), an array that holds information about the read sample(s) (`info`), the size of the arrays and the maximum number of samples to read. Every read sample in the samples array has related information in the info array at the same index.

```
ret = dds_read (reader, samples, info, MAX_SAMPLES, MAX_SAMPLES);
```

The `dds_read` function returns the number of samples it actually read. We can use that to determine if the function actually read some data. When it has, then it is still possible that the data part of the sample is not valid. This has some use cases when there is no real data, but still the state of the related sample has changed (for instance it was deleted). This will normally not happen in the *Hello World!* example. But we check for it anyway.

```
if ((ret > 0) && (info[0].valid_data))
```

If data has been read, then we can cast the void pointer to the actual message data type and display the contents. The polling loop is quit as well in this case.

```
msg = (HelloWorldData_Msg*) samples[0];
printf ("=== [Subscriber] Received : ");
printf ("Message (%d, %s)\n", msg->userID, msg->message);
break;
```

When data is received and the polling loop is stopped, we need to clean up.

```
HelloWorldData_Msg_free (samples[0], DDS_FREE_ALL);
dds_delete (participant);
```

All the entities that are created using the participant are also deleted. This means that deleting the participant will automatically delete the topic and reader as well.

3.3.2 *Hello World!* Publisher Source Code

Publisher.c contains the source that will write an *Hello World!* message on which the subscriber is waiting.

```
1  #include "ddsc/dds.h"
2  #include "HelloWorldData.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main (int argc, char ** argv)
7  {
8      dds_entity_t participant;
9      dds_entity_t topic;
10     dds_entity_t writer;
11     dds_return_t ret;
12     HelloWorldData_Msg msg;
13     (void)argc;
14     (void)argv;
15
16     /* Create a Participant. */
17     participant = dds_create_participant (DDS_DOMAIN_DEFAULT, NULL, NULL);
18     DDS_ERR_CHECK (participant, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
19
20     /* Create a Topic. */
21     topic = dds_create_topic (participant, &HelloWorldData_Msg_desc,
22                               "HelloWorldData_Msg", NULL, NULL);
23     DDS_ERR_CHECK (topic, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
```

(continues on next page)

(continued from previous page)

```
24
25      /* Create a Writer. */
26      writer = dds_create_writer (participant, topic, NULL, NULL);
27
28      printf("=== [Publisher]  Waiting for a reader to be discovered ...\n");
29
30      ret = dds_set_status_mask(writer, DDS_PUBLICATION_MATCHED_STATUS);
31      DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
32
33      while(true)
34      {
35        uint32_t status;
36        ret = dds_get_status_changes (writer, &status);
37        DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
38
39        if (status == DDS_PUBLICATION_MATCHED_STATUS) {
40          break;
41        }
42        /* Polling sleep. */
43        dds_sleepfor (DDS_MSECS (20));
44      }
45
46      /* Create a message to write. */
47      msg.userID = 1;
48      msg.message = "Hello World";
49
50      printf ("=== [Publisher]  Writing : ");
51      printf ("Message (%d, %s)\n", msg.userID, msg.message);
52
53      ret = dds_write (writer, &msg);
54      DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
55
56      /* Deleting the participant will delete all its children recursively as well. */
57      ret = dds_delete (participant);
58      DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);
59
60      return EXIT_SUCCESS;
61  }
```

We will be using the DDS API and the *HelloWorldData_Msg* type to sent data. For that, we need to include the appropriate header files.

```
#include "ddsc/dds.h"
#include "HelloWorldData.h"
```

Just like with the *reader in subscriber.c*, we need a participant and a topic to be able to create a writer. We use the same topic name as in subscriber.c. Otherwise the reader and writer are not considered related and data will not be sent between them.

```
dds_entity_t participant;
dds_entity_t topic;
dds_entity_t writer;

participant = dds_create_participant (DDS_DOMAIN_DEFAULT, NULL, NULL);
topic = dds_create_topic (participant, &HelloWorldData_Msg_desc,
                          "HelloWorldData_Msg", NULL, NULL);
writer = dds_create_writer (participant, topic, NULL, NULL);
```

The DDS middleware is a publication/subscription implementation. This means that it will discover related readers and writers (i.e. readers and writers sharing the same data type and topic name) and connect them so that written data can be received by readers without the application having to worry about it. There is a catch though: this discovery and coupling takes a small amount of time. There are various ways to work around this problem. The following can be done to properly connect readers and writers:

- Wait for the publication/subscription matched events
  - The Subscriber should wait for a subscription matched event
  - The Publisher should wait for a publication matched event.

  The use of these events will be outside the scope of this example
- Poll for the publication/subscription matches statusses
  - The Subscriber should poll for a subscription matched status to be set
  - The Publisher should poll for a publication matched status to be set

  The Publisher in this example uses the polling schema.
- Let the publisher sleep for a second before writing a sample. This is not recommended since a second may not be enough on several networks
- Accept that the reader miss a few samples at startup. This may be acceptable in cases where the publishing rate is high enough.

As said, the publisher of this example polls for the publication matched status. To make this happen, the writer must be instructed to 'listen' for this status. The following line of code makes sure the writer does so.

```
dds_set_status_mask(writer, DDS_PUBLICATION_MATCHED_STATUS);
```

Now the polling may start:

```
while(true)
{
    uint32_t status;
    ret = dds_get_status_changes (writer, &status);
    DDS_ERR_CHECK(ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT);

    if (status == DDS_PUBLICATION_MATCHED_STATUS) {
        break;
    }
    /* Polling sleep. */
    dds_sleepfor (DDS_MSECS (20));
}
```

After this loop, we are sure that a matching reader has been started. Now, we commence to writing the data. First the data must be initialized

```
HelloWorldData_Msg msg;

msg.userID = 1;
msg.message = "Hello World";
```

Then we can actually sent the message to be received by the subscriber.

```
ret = dds_write (writer, &msg);
```

After the sample is written, we need to clean up.

```
ret = dds_delete (participant);
```

All the entities that are created using the participant are also deleted. This means that deleting the participant will automatically delete the topic and writer as well.

CHAPTER 4

What's next?

Want to know more about DDS? The primary source of information is the OMG website at http://www.omg.org and specifically the DDS Getting Started page and the DDS specification itself. The specification is a bit wordy and of course deals with minute details, but it is surprisingly easy to follow for a specification.

There are also various resources on the web dealing with DDS in general, as the various vendors have posted tutorials, presentations, general information and documentation on their products. While the details between the various implementations do differ, they have much more in common than what separates them, and so this information is also applicable to Eclipse Eclipse Cyclone DDS. The one thing in which Eclipse Cyclone DDS really differs is in the details of API, but that's just syntax.

Obviously there are also things specific to Eclipse Cyclone DDS. The level of documentation of Eclipse is not nearly what it should be, but that should improve over time.

And last but note least: please always feel welcome to ask questions on GitHub!

CHAPTER 5

Uninstalling Eclipse Cyclone DDS

TBD.

CHAPTER 6

Eclipse Cyclone DDS C API Reference

struct dds_aligned_allocator

Public Members

void *(***alloc**) (size_t size, size_t align)

void (***free**) (size_t size, void *ptr)

struct dds_allocator

Public Members

void *(***malloc**) (size_t size)

void *(***realloc**) (void *ptr, size_t size)

void (***free**) (void *ptr)

struct dds_builtintopic_endpoint

Public Members

dds_builtintopic_guid_t **key**

dds_builtintopic_guid_t **participant_key**

char ***topic_name**

char ***type_name**

dds_qos_t ***qos**

struct dds_builtintopic_guid

Public Members

uint8_t **v**[16]

struct dds_builtintopic_participant

Public Members

dds_builtintopic_guid_t **key**

dds_qos_t ***qos**

struct dds_history_qospolicy
#include <*dds_public_qos.h*> History QoS: Applies to Topic, DataReader, DataWriter

Public Members

dds_history_kind_t **kind**

int32_t **depth**

struct dds_inconsistent_topic_status
#include <*dds_public_status.h*> DCPS_Status_InconsistentTopic

Public Members

uint32_t **total_count**

int32_t **total_count_change**

struct dds_key_descriptor

Public Members

const char ***m_name**

uint32_t **m_index**

struct dds_liveliness_changed_status
#include <*dds_public_status.h*> DCPS_Status_LivelinessChanged

Public Members

uint32_t **alive_count**

uint32_t **not_alive_count**

int32_t **alive_count_change**

int32_t **not_alive_count_change**

dds_instance_handle_t **last_publication_handle**

struct dds_liveliness_lost_status
#include <*dds_public_status.h*> DCPS_Status_LivelinessLost

Public Members

uint32_t **total_count**

int32_t **total_count_change**

struct dds_offered_deadline_missed_status
#include <*dds_public_status.h*> DCPS_Status_OfferedDeadlineMissed

Public Members

uint32_t **total_count**

int32_t **total_count_change**

dds_instance_handle_t **last_instance_handle**

struct dds_offered_incompatible_qos_status
#include <*dds_public_status.h*> DCPS_Status_OfferedIncompatibleQoS

Public Members

uint32_t **total_count**

int32_t **total_count_change**

uint32_t **last_policy_id**

struct dds_publication_matched_status
#include <*dds_public_status.h*> DCPS_Status_PublicationMatched

Public Members

uint32_t **total_count**

int32_t **total_count_change**

uint32_t **current_count**

int32_t **current_count_change**

dds_instance_handle_t **last_subscription_handle**

struct dds_requested_deadline_missed_status
#include <*dds_public_status.h*> DCPS_Status_RequestedDeadlineMissed

Public Members

uint32_t **total_count**

int32_t **total_count_change**

dds_instance_handle_t **last_instance_handle**

struct dds_requested_incompatible_qos_status
#include <*dds_public_status.h*> DCPS_Status_RequestedIncompatibleQoS

Public Members

uint32_t **total_count**

int32_t **total_count_change**

uint32_t **last_policy_id**

struct dds_resource_limits_qospolicy
#include <dds_public_qos.h> ResourceLimits QoS: Applies to Topic, DataReader, DataWriter

Public Members

int32_t **max_samples**

int32_t **max_instances**

int32_t **max_samples_per_instance**

struct dds_sample_info
#include <dds.h> Contains information about the associated data value

Public Members

dds_sample_state_t **sample_state**
Sample state

dds_view_state_t **view_state**
View state

dds_instance_state_t **instance_state**
Instance state

bool **valid_data**
Indicates whether there is a data associated with a sample

- true, indicates the data is valid
- false, indicates the data is invalid, no data to read

dds_time_t **source_timestamp**
timestamp of a data instance when it is written

dds_instance_handle_t **instance_handle**
handle to the data instance

dds_instance_handle_t **publication_handle**
handle to the publisher

uint32_t **disposed_generation_count**
count of instance state change from NOT_ALIVE_DISPOSED to ALIVE

uint32_t **no_writers_generation_count**
count of instance state change from NOT_ALIVE_NO_WRITERS to ALIVE

uint32_t **sample_rank**
indicates the number of samples of the same instance that follow the current one in the collection

uint32_t **generation_rank**
difference in generations between the sample and most recent sample of the same instance that appears in the returned collection

uint32_t **absolute_generation_rank**
difference in generations between the sample and most recent sample of the same instance when read/take was called

struct dds_sample_lost_status
#include <dds_public_status.h> DCPS_Status_SampleLost

Public Members

uint32_t **total_count**

int32_t **total_count_change**

struct dds_sample_rejected_status
#include <dds_public_status.h> DCPS_Status_SampleRejected

Public Members

uint32_t **total_count**

int32_t **total_count_change**

dds_sample_rejected_status_kind **last_reason**

dds_instance_handle_t **last_instance_handle**

struct dds_sequence

Public Members

uint32_t **_maximum**

uint32_t **_length**

uint8_t ***_buffer**

bool **_release**

struct dds_stream

Public Members

dds_uptr_t **m_buffer**

uint32_t **m_size**

uint32_t **m_index**

bool **m_endian**

bool **m_failed**

struct dds_subscription_matched_status
#include <dds_public_status.h> DCPS_Status_SubscriptionMatched

Public Members

uint32_t **total_count**

int32_t **total_count_change**

uint32_t **current_count**

int32_t **current_count_change**

dds_instance_handle_t **last_publication_handle**

struct dds_topic_descriptor

Public Members

const uint32_t **m_size**

const uint32_t **m_align**

const uint32_t **m_flagset**

const uint32_t **m_nkeys**

const char ***m_typename**

const *dds_key_descriptor_t* ***m_keys**

const uint32_t **m_nops**

const uint32_t ***m_ops**

const char ***m_meta**

union dds_uptr_t

Public Members

uint8_t ***p8**

uint16_t ***p16**

uint32_t ***p32**

uint64_t ***p64**

float ***pf**

double ***pd**

void ***pv**

file **dds.h**

#include "os/os_public.h"#include "ddsc/dds_export.h"#include "ddsc/dds_public_stream.h"#include "ddsc/dds_public_impl.h"#include "ddsc/dds_public_alloc.h"#include "ddsc/dds_public_time.h"#include "ddsc/dds_public_qos.h"#include "ddsc/dds_public_error.h"#include "ddsc/dds_public_status.h"#include "ddsc/dds_public_listener.h" Eclipse Cyclone DDS C header.

Communication Status definitions

DDS_INCONSISTENT_TOPIC_STATUS

DDS_OFFERED_DEADLINE_MISSED_STATUS
The deadline that the writer has committed through its deadline QoS policy was not respected for a specific instance.

DDS_REQUESTED_DEADLINE_MISSED_STATUS
The deadline that the reader was expecting through its deadline QoS policy was not respected for a specific instance.

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS
A QoS policy setting was incompatible with what was requested.

DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS
A QoS policy setting was incompatible with what is offered.

DDS_SAMPLE_LOST_STATUS
A sample has been lost (never received).

DDS_SAMPLE_REJECTED_STATUS
A (received) sample has been rejected.

DDS_DATA_ON_READERS_STATUS
New information is available.

DDS_DATA_AVAILABLE_STATUS
New information is available.

DDS_LIVELINESS_LOST_STATUS
The liveliness that the DDS_DataWriter has committed through its liveliness QoS policy was not respected; thus readers will consider the writer as no longer "alive".

DDS_LIVELINESS_CHANGED_STATUS
The liveliness of one or more writers, that were writing instances read through the readers has changed. Some writers have become "alive" or "not alive".

DDS_PUBLICATION_MATCHED_STATUS
The writer has found a reader that matches the topic and has a compatible QoS.

DDS_SUBSCRIPTION_MATCHED_STATUS
The reader has found a writer that matches the topic and has a compatible QoS.

enum dds_status_id
Another topic exists with the same name but with different characteristics.

Values:

DDS_INCONSISTENT_TOPIC_STATUS_ID

DDS_OFFERED_DEADLINE_MISSED_STATUS_ID

DDS_REQUESTED_DEADLINE_MISSED_STATUS_ID

DDS_OFFERED_INCOMPATIBLE_QOS_STATUS_ID

DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS_ID

DDS_SAMPLE_LOST_STATUS_ID

DDS_SAMPLE_REJECTED_STATUS_ID

DDS_DATA_ON_READERS_STATUS_ID

DDS_DATA_AVAILABLE_STATUS_ID

DDS_LIVELINESS_LOST_STATUS_ID

DDS_LIVELINESS_CHANGED_STATUS_ID

DDS_PUBLICATION_MATCHED_STATUS_ID

DDS_SUBSCRIPTION_MATCHED_STATUS_ID

typedef enum *dds_status_id* **dds_status_id_t**
Another topic exists with the same name but with different characteristics.

Typedefs

typedef int32_t **dds_return_t**
Return code indicating success (DDS_RETCODE_OK) or failure. If a given operation failed the value will be a unique error code and *dds_err_nr()* must be used to extract the DDS_RETCODE_* value.

typedef int32_t **dds_entity_t**
Handle to an entity. A valid entity handle will always have a positive integer value. Should the value be negative, the value represents a unique error code. *dds_err_nr()* can be used to extract the DDS_RETCODE_* value.

typedef enum *dds_sample_state* **dds_sample_state_t**
Read state for a data value

typedef enum *dds_view_state* **dds_view_state_t**
View state of an instance relative to the samples

typedef enum *dds_instance_state* **dds_instance_state_t**
Defines the state of the instance

typedef struct *dds_sample_info* **dds_sample_info_t**
Contains information about the associated data value

typedef struct *dds_builtintopic_guid* **dds_builtintopic_guid_t**

typedef struct *dds_builtintopic_participant* **dds_builtintopic_participant_t**

typedef struct *dds_builtintopic_endpoint* **dds_builtintopic_endpoint_t**

typedef bool (***dds_topic_filter_fn**)(**const** void *sample)
Topic filter function

typedef bool (***dds_querycondition_filter_fn**)(**const** void *sample)

typedef intptr_t **dds_attach_t**
Waitset attachment argument.

Every entity that is attached to the waitset can be accompanied by such an attachment argument. When the waitset wait is unblocked because of an entity that triggered, then the returning array will be populated with these attachment arguments that are related to the triggered entity.

Enums

enum dds_sample_state
Read state for a data value

Values:

DDS_SST_READ = DDS_READ_SAMPLE_STATE
DataReader has already accessed the sample by read

DDS_SST_NOT_READ = DDS_NOT_READ_SAMPLE_STATE
DataReader has not accessed the sample before

enum dds_view_state
View state of an instance relative to the samples

Values:

DDS_VST_NEW = DDS_NEW_VIEW_STATE
DataReader is accessing the sample for the first time when the instance is alive

DDS_VST_OLD = DDS_NOT_NEW_VIEW_STATE
DataReader accessed the sample before

enum dds_instance_state
Defines the state of the instance

Values:

DDS_IST_ALIVE = DDS_ALIVE_INSTANCE_STATE
Samples received for the instance from the live data writers

DDS_IST_NOT_ALIVE_DISPOSED = DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE
Instance was explicitly disposed by the data writer

DDS_IST_NOT_ALIVE_NO_WRITERS = DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE
Instance has been declared as not alive by data reader as there are no live data writers writing that instance

Functions

dds_domainid_t **dds_domain_default** (void)
Returns the default domain identifier.

The default domain identifier can be configured in the configuration file or be set through an evironment variable ({DDSC_PROJECT_NAME_NOSPACE_CAPS}_DOMAIN).

Return Default domain identifier

dds_return_t **dds_enable** (*dds_entity_t entity*)
Enable entity.

This operation enables the dds_entity_t. Created dds_entity_t objects can start in either an enabled or disabled state. This is controlled by the value of the entityfactory policy on the corresponding parent entity for the given entity. Enabled entities are immediately activated at creation time meaning all their immutable QoS settings can no longer be changed. Disabled Entities are not yet activated, so it is still possible to change their immutable QoS settings. However, once activated the immutable QoS settings can no longer be changed. Creating disabled entities can make sense when the creator of the DDS_Entity does not yet know which QoS settings to apply, thus allowing another piece of code to set the QoS later on.

Note Delayed entity enabling is not supported yet (CHAM-96).

The default setting of DDS_EntityFactoryQosPolicy is such that, by default, entities are created in an enabled state so that it is not necessary to explicitly call dds_enable on newly-created entities.

The dds_enable operation produces the same results no matter how many times it is performed. Calling dds_enable on an already enabled DDS_Entity returns DDS_RETCODE_OK and has no effect.

If an Entity has not yet been enabled, the only operations that can be invoked on it are: the ones to set, get or copy the QosPolicy settings, the ones that set (or get) the Listener, the ones that get the Status and the dds_get_status_changes operation (although the status of a disabled entity never changes). Other operations will return the error DDS_RETCODE_NOT_ENABLED.

Entities created with a parent that is disabled, are created disabled regardless of the setting of the entity-factory policy.

If the entityfactory policy has autoenable_created_entities set to TRUE, the dds_enable operation on the parent will automatically enable all child entities created with the parent.

The Listeners associated with an Entity are not called until the Entity is enabled. Conditions associated with an Entity that is not enabled are "inactive", that is, have a trigger_value which is FALSE.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: The entity to enable.

Return Value

- `DDS_RETCODE_OK`: The listeners of to the entity have been successfully been copied into the specified listener parameter.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The parent of the given Entity is not enabled.

dds_return_t **dds_delete** (*dds_entity_t entity*)

Delete given entity.

This operation will delete the given entity. It will also automatically delete all its children, childrens' children, etc entities.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity to delete.

Return Value

- `DDS_RETCODE_OK`: The entity and its children (recursive are deleted).
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_get_publisher** (*dds_entity_t writer*)

Get entity publisher.

This operation returns the publisher to which the given entity belongs. For instance, it will return the Publisher that was used when creating a DataWriter (when that DataWriter was provided here).

Return A valid entity or an error code.

Parameters

- `entity`: Entity from which to get its publisher.

Return Value

- `>0`: A valid publisher handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_get_subscriber** (*dds_entity_t entity*)
Get entity subscriber.

This operation returns the subscriber to which the given entity belongs. For instance, it will return the Subscriber that was used when creating a DataReader (when that DataReader was provided here).

Return A valid subscriber handle or an error code.

Parameters

- `entity`: Entity from which to get its subscriber.

Return Value

- `>0`: A valid subscriber handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_get_datareader** (*dds_entity_t condition*)
Get entity datareader.

This operation returns the datareader to which the given entity belongs. For instance, it will return the DataReader that was used when creating a ReadCondition (when that ReadCondition was provided here).

Return A valid reader handle or an error code.

Parameters

- `entity`: Entity from which to get its datareader.

Return Value

- `>0`: A valid reader handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_mask** (*dds_entity_t condition*, uint32_t **mask*)
Get the mask of a condition.

This operation returns the mask that was used to create the given condition.

Return A dds_return_t indicating success or failure.

Parameters

- `condition`: Read or Query condition that has a mask.

Return Value

- `DDS_RETCODE_OK`: Success (given mask is set).
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The mask arg is NULL.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_instance_handle** (*dds_entity_t entity*, *dds_instance_handle_t *ihdl*)
Returns the instance handle that represents the entity.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity of which to get the instance handle.
- `ihdl`: Pointer to dds_instance_handle_t.

Return Value

- `DDS_RETCODE_OK`: Success.
- `DDS_RETCODE_ERROR`: An internal error has occurred.

dds_return_t **dds_read_status** (*dds_entity_t entity*, uint32_t *status, uint32_t *mask*)
Read the status set for the entity.

This operation reads the status(es) set for the entity based on the enabled status and mask set. It does not clear the read status(es).

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity on which the status has to be read.
- `status`: Returns the status set on the entity, based on the enabled status.
- `mask`: Filter the status condition to be read (can be NULL).

Return Value

- `DDS_RETCODE_OK`: Success.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_take_status** (*dds_entity_t entity*, uint32_t *status, uint32_t *mask*)
Read the status set for the entity.

This operation reads the status(es) set for the entity based on the enabled status and mask set. It clears the status set after reading.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity on which the status has to be read.

- `status`: Returns the status set on the entity, based on the enabled status.
- `mask`: Filter the status condition to be read (can be NULL).

Return Value

- `DDS_RETCODE_OK`: Success.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_status_changes** (*dds_entity_t entity*, uint32_t **status*)
Get changed status(es)

This operation returns the status changes since they were last read.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity on which the statuses are read.
- `status`: Returns the current set of triggered statuses.

Return Value

- `DDS_RETCODE_OK`: Success.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_status_mask** (*dds_entity_t entity*, uint32_t **mask*)
Get enabled status on entity.

This operation returns the status enabled on the entity

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity to get the status.
- `status`: Status set on the entity.

Return Value

- `DDS_RETCODE_OK`: Success.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_enabled_status** (*dds_entity_t entity*, uint32_t **mask*)

dds_return_t **dds_set_status_mask** (*dds_entity_t entity*, uint32_t *mask*)
Set status enabled on entity.

This operation enables the status(es) based on the mask set

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity to enable the status.
- `mask`: Status value that indicates the status to be enabled.

Return Value

- `DDS_RETCODE_OK`: Success.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_set_enabled_status** (*dds_entity_t entity*, uint32_t *mask*)

dds_return_t **dds_get_qos** (*dds_entity_t entity*, *dds_qos_t *qos*)
Get entity QoS policies.

This operation allows access to the existing set of QoS policies for the entity.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity on which to get qos.
- `qos`: Pointer to the qos structure that returns the set policies.

Return Value

- `DDS_RETCODE_OK`: The existing set of QoS policy values applied to the entity has successfully been copied into the specified qos parameter.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The qos parameter is NULL.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_set_qos** (*dds_entity_t entity*, **const** *dds_qos_t *qos*)
Set entity QoS policies.

This operation replaces the existing set of Qos Policy settings for an entity. The parameter qos must contain the struct with the QosPolicy settings which is checked for self-consistency.

The set of QosPolicy settings specified by the qos parameter are applied on top of the existing QoS, replacing the values of any policies previously set (provided, the operation returned DDS_RETCODE_OK).

Not all policies are changeable when the entity is enabled.

Note Currently only Latency Budget and Ownership Strength are changeable QoS that can be set.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity from which to get qos.
- `qos`: Pointer to the qos structure that provides the policies.

Return Value

- `DDS_RETCODE_OK`: The new QoS policies are set.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The qos parameter is NULL.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_IMMUTABLE_POLICY`: The entity is enabled and one or more of the policies of the QoS are immutable.
- `DDS_RETCODE_INCONSISTENT_POLICY`: A few policies within the QoS are not consistent with each other.

dds_return_t **dds_get_listener** (*dds_entity_t entity*, *dds_listener_t *listener*)

Get entity listeners.

This operation allows access to the existing listeners attached to the entity.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity on which to get the listeners.
- `listener`: Pointer to the listener structure that returns the set of listener callbacks.

Return Value

- `DDS_RETCODE_OK`: The listeners of to the entity have been successfully been copied into the specified listener parameter.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The listener parameter is NULL.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_set_listener** (*dds_entity_t entity*, **const** *dds_listener_t *listener*)

Set entity listeners.

This operation attaches a dds_listener_t to the dds_entity_t. Only one Listener can be attached to each Entity. If a Listener was already attached, this operation will replace it with the new one. In other words, all related callbacks are replaced (possibly with NULL).

When listener parameter is NULL, all listener callbacks that were possibly set on the Entity will be removed.

For each communication status, the StatusChangedFlag flag is initially set to FALSE. It becomes TRUE whenever that plain communication status changes. For each plain communication status activated in the mask, the associated Listener callback is invoked and the communication status is reset to FALSE, as the listener implicitly accesses the status which is passed as a parameter to that operation. The status is reset prior to calling the listener, so if the application calls the get_<status_name> from inside the listener it will see the status already reset.

Note Not all listener callbacks are related to all entities.

In case a related callback within the Listener is not set, the Listener of the Parent entity is called recursively, until a Listener with the appropriate callback set has been found and called. This allows the application to set (for instance) a default behaviour in the Listener of the containing Publisher and a DataWriter specific behaviour when needed. In case the callback is not set in the Publishers' Listener either, the communication status will be propagated to the Listener of the DomainParticipant of the containing DomainParticipant. In case the callback is not set in the DomainParticipants' Listener either, the Communication Status flag will be set, resulting in a possible WaitSet trigger.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity on which to get the listeners.
- `listener`: Pointer to the listener structure that contains the set of listener callbacks (maybe NULL).

Return Value

- `DDS_RETCODE_OK`: The listeners of to the entity have been successfully been copied into the specified listener parameter.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_create_participant** (**const** *dds_domainid_t domain*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*)

Creates a new instance of a DDS participant in a domain.

If domain is set (not DDS_DOMAIN_DEFAULT) then it must match if the domain has also been configured or an error status will be returned. Currently only a single domain can be configured by providing configuration file. If no configuration file exists, the default domain is configured as 0.

Return A valid participant handle or an error code.

Parameters

- `domain`: The domain in which to create the participant (can be DDS_DOMAIN_DEFAULT). Valid values for domain id are between 0 and 230. DDS_DOMAIN_DEFAULT is for using the domain in the configuration.
- `qos`: The QoS to set on the new participant (can be NULL).
- `listener`: Any listener functions associated with the new participant (can be NULL).

Return Value

- `>0`: A valid participant handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.

dds_entity_t **dds_get_parent** (*dds_entity_t entity*)

Get entity parent.

This operation returns the parent to which the given entity belongs. For instance, it will return the Participant that was used when creating a Publisher (when that Publisher was provided here).

When a reader or a writer are created with a partition, then a subscriber or publisher respectively are created implicitly. These implicit subscribers or publishers will be deleted automatically when the reader or writer is deleted. However, when this function returns such an implicit entity, it is from there on out considered

'explicit'. This means that it isn't deleted automatically anymore. The application should explicitly call dds_delete on those entities now (or delete the parent participant which will delete all entities within its hierarchy).

Return A valid entity handle or an error code.

Parameters

- `entity`: Entity from which to get its parent.

Return Value

- `>0`: A valid entity handle.
- `DDS_ENTITY_NIL`: Called with a participant.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_get_participant** (*dds_entity_t entity*)
Get entity participant.

This operation returns the participant to which the given entity belongs. For instance, it will return the Participant that was used when creating a Publisher that was used to create a DataWriter (when that DataWriter was provided here).

TODO: Link to generic dds entity relations documentation.

Return A valid entity or an error code.

Parameters

- `entity`: Entity from which to get its participant.

Return Value

- `>0`: A valid participant handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_children** (*dds_entity_t entity*, *dds_entity_t* **children*, size_t *size*)
Get entity children.

This operation returns the children that the entity contains. For instance, it will return all the Topics, Publishers and Subscribers of the Participant that was used to create those entities (when that Participant is provided here).

This functions takes a pre-allocated list to put the children in and will return the number of found children. It is possible that the given size of the list is not the same as the number of found children. If less children are found, then the last few entries in the list are untouched. When more children are found, then only 'size' number of entries are inserted into the list, but still complete count of the found children is returned. Which children are returned in the latter case is undefined.

When supplying NULL as list and 0 as size, you can use this to acquire the number of children without having to pre-allocate a list.

When a reader or a writer are created with a partition, then a subscriber or publisher respectively are created implicitly. These implicit subscribers or publishers will be deleted automatically when the reader or writer is deleted. However, when this function returns such an implicit entity, it is from there on out considered 'explicit'. This means that it isn't deleted automatically anymore. The application should explicitly call dds_delete on those entities now (or delete the parent participant which will delete all entities within its hierarchy).

Return Number of children or an error code.

Parameters

- `entity`: Entity from which to get its children.
- `children`: Pre-allocated array to contain the found children.
- `size`: Size of the pre-allocated children's list.

Return Value

- `>=0`: Number of childer found children (can be larger than 'size').
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The children parameter is NULL, while a size is provided.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_domainid** (*dds_entity_t entity*, *dds_domainid_t *id*)

Get the domain id to which this entity is attached.

When creating a participant entity, it is attached to a certain domain. All the children (like Publishers) and childrens' children (like DataReaders), etc are also attached to that domain.

This function will return the original domain ID when called on any of the entities within that hierarchy.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Entity from which to get its children.
- `id`: Pointer to put the domain ID in.

Return Value

- `DDS_RETCODE_OK`: Domain ID was returned.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The id parameter is NULL.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_lookup_participant** (*dds_domainid_t domain_id*, *dds_entity_t *participants*, size_t *size*)

Get participants of a domain.

This operation acquires the participants created on a domain and returns the number of found participants.

This function takes a domain id with the size of pre-allocated participant's list in and will return the number of found participants. It is possible that the given size of the list is not the same as the number of found

participants. If less participants are found, then the last few entries in an array stay untouched. If more participants are found and the array is too small, then the participants returned are undefined.

Return Number of participants found or and error code.

Parameters

- `domain_id`: The domain id.
- `participants`: The participant for domain.
- `size`: Size of the pre-allocated participant's list.

Return Value

- `>0`: Number of participants found.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The participant parameter is NULL, while a size is provided.

dds_entity_t **dds_create_topic** (*dds_entity_t participant*, **const** *dds_topic_descriptor_t *descriptor*, **const** char **name*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*)

Creates a new topic with default type handling.

The type name for the topic is taken from the generated descriptor. Topic matching is done on a combination of topic name and type name.

Return A valid topic handle or an error code.

Parameters

- `participant`: Participant on which to create the topic.
- `descriptor`: An IDL generated topic descriptor.
- `name`: Name of the topic.
- `qos`: QoS to set on the new topic (can be NULL).
- `listener`: Any listener functions associated with the new topic (can be NULL).

Return Value

- `>=0`: A valid topic handle.
- `DDS_RETCODE_BAD_PARAMETER`: Either participant, descriptor, name or qos is invalid.

dds_entity_t **dds_create_topic_arbitrary** (*dds_entity_t participant*, **struct** ddsi_sertopic **sertopic*, **const** char **name*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*, **const** **struct** nn_plist **sedp_plist*)

dds_entity_t **dds_find_topic** (*dds_entity_t participant*, **const** char **name*)

Finds a named topic.

The returned topic should be released with dds_delete.

Return A valid topic handle or an error code.

Parameters

- `participant`: The participant on which to find the topic.

- `name`: The name of the topic to find.

Return Value

- `>0`: A valid topic handle.
- `DDS_RETCODE_BAD_PARAMETER`: Participant was invalid.

dds_return_t **dds_get_name** (*dds_entity_t topic*, char **name*, size_t *size*)
Returns the name of a given topic.

Return A dds_return_t indicating success or failure.

Parameters

- `topic`: The topic.
- `name`: Buffer to write the topic name to.
- `size`: Number of bytes available in the buffer.

Return Value

- `DDS_RETCODE_OK`: Success.

dds_return_t **dds_get_type_name** (*dds_entity_t topic*, char **name*, size_t *size*)
Returns the type name of a given topic.

Return A dds_return_t indicating success or failure.

Return DDS_RETCODE_OK Success.

Parameters

- `topic`: The topic.
- `name`: Buffer to write the topic type name to.
- `size`: Number of bytes available in the buffer.

void **dds_set_topic_filter** (*dds_entity_t topic*, *dds_topic_filter_fn filter*)
Sets a filter on a topic.

Parameters

- `topic`: The topic on which the content filter is set.
- `filter`: The filter function used to filter topic samples.

void **dds_topic_set_filter** (*dds_entity_t topic*, *dds_topic_filter_fn filter*)

dds_topic_filter_fn **dds_get_topic_filter** (*dds_entity_t topic*)
Gets the filter for a topic.

Return The topic filter.

Parameters

- `topic`: The topic from which to get the filter.

dds_topic_filter_fn **dds_topic_get_filter** (*dds_entity_t topic*)

dds_entity_t **dds_create_subscriber**(*dds_entity_t participant*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*)

Creates a new instance of a DDS subscriber.

Return A valid subscriber handle or an error code.

Parameters

- `participant`: The participant on which the subscriber is being created.
- `qos`: The QoS to set on the new subscriber (can be NULL).
- `listener`: Any listener functions associated with the new subscriber (can be NULL).

Return Value

- `>0`: A valid subscriber handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the parameters is invalid.

dds_entity_t **dds_create_publisher**(*dds_entity_t participant*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*)

Creates a new instance of a DDS publisher.

Return A valid publisher handle or an error code.

Parameters

- `participant`: The participant to create a publisher for.
- `qos`: The QoS to set on the new publisher (can be NULL).
- `listener`: Any listener functions associated with the new publisher (can be NULL).

Return Value

- `>0`: A valid publisher handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.

dds_return_t **dds_suspend**(*dds_entity_t publisher*)

Suspends the publications of the Publisher.

This operation is a hint to the Service so it can optimize its performance by e.g., collecting modifications to DDS writers and then batching them. The Service is not required to use the hint.

Every invocation of this operation must be matched by a corresponding call to

See *dds_resume* indicating that the set of modifications has completed.

Return A dds_return_t indicating success or failure.

Parameters

- `publisher`: The publisher for which all publications will be suspended.

Return Value

- `DDS_RETCODE_OK`: Publications suspended successfully.
- `DDS_RETCODE_BAD_PARAMETER`: The pub parameter is not a valid publisher.
- `DDS_RETCODE_UNSUPPORTED`: Operation is not supported.

dds_return_t **dds_resume** (*dds_entity_t publisher*)
Resumes the publications of the Publisher.

This operation is a hint to the Service to indicate that the application has completed changes initiated by a previous *dds_suspend()*. The Service is not required to use the hint.

The call to resume_publications must match a previous call to

See suspend_publications.

Return A dds_return_t indicating success or failure.

Parameters

- `publisher`: The publisher for which all publications will be resumed.

Return Value

- `DDS_RETCODE_OK`: Publications resumed successfully.
- `DDS_RETCODE_BAD_PARAMETER`: The pub parameter is not a valid publisher.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: No previous matching *dds_suspend()*.
- `DDS_RETCODE_UNSUPPORTED`: Operation is not supported.

dds_return_t **dds_wait_for_acks** (*dds_entity_t publisher_or_writer*, *dds_duration_t timeout*)
Waits at most for the duration timeout for acks for data in the publisher or writer.

This operation blocks the calling thread until either all data written by the publisher or writer is acknowledged by all matched reliable reader entities, or else the duration specified by the timeout parameter elapses, whichever happens first.

Return A dds_return_t indicating success or failure.

Parameters

- `publisher_or_writer`: Publisher or writer whose acknowledgments must be waited for
- `timeout`: How long to wait for acknowledgments before time out

Return Value

- `DDS_RETCODE_OK`: All acknowledgments successfully received with the timeout.
- `DDS_RETCODE_BAD_PARAMETER`: The publisher_or_writer is not a valid publisher or writer.
- `DDS_RETCODE_TIMEOUT`: Timeout expired before all acknowledgments from reliable reader entities were received.
- `DDS_RETCODE_UNSUPPORTED`: Operation is not supported.

dds_entity_t **dds_create_reader** (*dds_entity_t participant_or_subscriber*, *dds_entity_t topic*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*)
Creates a new instance of a DDS reader.

This implicit subscriber will be deleted automatically when the created reader is deleted.

Return A valid reader handle or an error code.

Parameters

- `participant_or_subscriber`: The participant or subscriber on which the reader is being created.
- `topic`: The topic to read.

- `qos`: The QoS to set on the new reader (can be NULL).
- `listener`: Any listener functions associated with the new reader (can be NULL).

Return Value

- `>0`: A valid reader handle.
- `DDS_RETCODE_ERROR`: An internal error occurred.

int **dds_reader_wait_for_historical_data** (*dds_entity_t reader*, *dds_duration_t max_wait*)

Wait until reader receives all historic data.

The operation blocks the calling thread until either all "historical" data is received, or else the duration specified by the max_wait parameter elapses, whichever happens first. A return value of 0 indicates that all the "historical" data was received; a return value of TIMEOUT indicates that max_wait elapsed before all the data was received.

Return a status, 0 on success, TIMEOUT on timeout or a negative value to indicate error.

Parameters

- `reader`: The reader on which to wait for historical data.
- `max_wait`: How long to wait for historical data before time out.

dds_entity_t **dds_create_writer** (*dds_entity_t participant_or_publisher*, *dds_entity_t topic*, **const** *dds_qos_t *qos*, **const** *dds_listener_t *listener*)

Creates a new instance of a DDS writer.

This implicit publisher will be deleted automatically when the created writer is deleted.

Return A valid writer handle or an error code.

Return >0 A valid writer handle.

Return DDS_RETCODE_ERROR An internal error occurred.

Parameters

- `participant_or_publisher`: The participant or publisher on which the writer is being created.
- `topic`: The topic to write.
- `qos`: The QoS to set on the new writer (can be NULL).
- `listener`: Any listener functions associated with the new writer (can be NULL).

dds_return_t **dds_register_instance** (*dds_entity_t writer*, *dds_instance_handle_t *handle*, **const** void **data*)

Registers an instance.

This operation registers an instance with a key value to the data writer and returns an instance handle that could be used for successive write & dispose operations. When the handle is not allocated, the function will return and error and the handle will be un-touched.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to which instance has be associated.
- `handle`: The instance handle.

- `data`: The instance with the key value.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.

dds_return_t **dds_unregister_instance** (*dds_entity_t writer*, **const** void **data*)

Unregisters an instance.

This operation reverses the action of register instance, removes all information regarding the instance and unregisters an instance with a key value from the data writer.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to which instance is associated.
- `data`: The instance with the key value.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.

dds_return_t **dds_unregister_instance_ih** (*dds_entity_t writer*, *dds_instance_handle_t handle*)

Unregisters an instance.

This operation unregisters the instance which is identified by the key fields of the given typed instance handle.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to which instance is associated.
- `handle`: The instance handle.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.

dds_return_t **dds_unregister_instance_ts** (*dds_entity_t writer*, **const** void **data*, *dds_time_t timestamp*)

Unregisters an instance.

This operation reverses the action of register instance, removes all information regarding the instance and unregisters an instance with a key value from the data writer. It also provides a value for the timestamp explicitly.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to which instance is associated.
- `data`: The instance with the key value.
- `timestamp`: The timestamp used at registration.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.

dds_return_t **dds_unregister_instance_ih_ts** (*dds_entity_t writer*, *dds_instance_handle_t handle*, *dds_time_t timestamp*)

Unregisters an instance.

This operation unregisters an instance with a key value from the handle. Instance can be identified from instance handle. If an unregistered key ID is passed as an instance data, an error is logged and not flagged as return value.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to which instance is associated.
- `handle`: The instance handle.
- `timestamp`: The timestamp used at registration.

Return Value

- `DDS_RETCODE_OK`: The operation was successful
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object

dds_return_t **dds_writedispose** (*dds_entity_t writer*, **const** void **data*)

This operation modifies and disposes a data instance.

This operation requests the Data Distribution Service to modify the instance and mark it for deletion. Copies of the instance and its corresponding samples, which are stored in every connected reader and, dependent on the QoS policy settings (also in the Transient and Persistent stores) will be modified and marked for deletion by setting their dds_instance_state_t to DDS_IST_NOT_ALIVE_DISPOSED.

If the history QoS policy is set to DDS_HISTORY_KEEP_ALL, the dds_writedispose operation on the writer may block if the modification would cause data to be lost because one of the limits, specified in the resource_limits QoS policy, to be exceeded. In case the synchronous attribute value of the reliability Qos policy is set to true for communicating writers and readers then the writer will wait until all synchronous readers have acknowledged the data. Under these circumstances, the max_blocking_time attribute of the reliability QoS policy configures the maximum time the dds_writedispose operation may block. If max_blocking_time elapses before the writer is able to store the modification without exceeding the limits and all expected acknowledgements are received, the dds_writedispose operation will fail and returns DDS_RETCODE_TIMEOUT.

Return A dds_return_t indicating success or failure.

Parameters

- writer: The writer to dispose the data instance from.
- data: The data to be written and disposed.

Return Value

- DDS_RETCODE_OK: The sample is written and the instance is marked for deletion.
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: At least one of the arguments is invalid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.
- DDS_RETCODE_TIMEOUT: Either the current action overflowed the available resources as specified by the combination of the reliability QoS policy, history QoS policy and resource_limits QoS policy, or the current action was waiting for data delivery acknowledgement by synchronous readers. This caused blocking of this operation, which could not be resolved before max_blocking_time of the reliability QoS policy elapsed.

dds_return_t **dds_writedispose_ts**(*dds_entity_t writer*, **const** void **data*, *dds_time_t timestamp*)

This operation modifies and disposes a data instance with a specific timestamp.

This operation performs the same functions as dds_writedispose except that the application provides the value for the source_timestamp that is made available to connected reader objects. This timestamp is important for the interpretation of the destination_order QoS policy.

Return A dds_return_t indicating success or failure.

Parameters

- writer: The writer to dispose the data instance from.
- data: The data to be written and disposed.
- timestamp: The timestamp used as source timestamp.

Return Value

- DDS_RETCODE_OK: The sample is written and the instance is marked for deletion.
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: At least one of the arguments is invalid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.
- DDS_RETCODE_TIMEOUT: Either the current action overflowed the available resources as specified by the combination of the reliability QoS policy, history QoS policy and resource_limits QoS policy, or the current action was waiting for data delivery acknowledgement by synchronous readers. This caused blocking of this operation, which could not be resolved before max_blocking_time of the reliability QoS policy elapsed.

dds_return_t **dds_dispose**(*dds_entity_t writer*, **const** void **data*)

This operation disposes an instance, identified by the data sample.

This operation requests the Data Distribution Service to modify the instance and mark it for deletion. Copies of the instance and its corresponding samples, which are stored in every connected reader and, dependent on the QoS policy settings (also in the Transient and Persistent stores) will be modified and marked for deletion by setting their dds_instance_state_t to DDS_IST_NOT_ALIVE_DISPOSED.

If the history QoS policy is set to DDS_HISTORY_KEEP_ALL, the dds_writedispose operation on the writer may block if the modification would cause data to be lost because one of the limits, specified in the resource_limits QoS policy, to be exceeded. In case the synchronous attribute value of the reliability Qos policy is set to true for communicating writers and readers then the writer will wait until all synchronous readers have acknowledged the data. Under these circumstances, the max_blocking_time attribute of the reliability QoS policy configures the maximum time the dds_writedispose operation may block. If max_blocking_time elapses before the writer is able to store the modification without exceeding the limits and all expected acknowledgements are received, the dds_writedispose operation will fail and returns DDS_RETCODE_TIMEOUT.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to dispose the data instance from.
- `data`: The data sample that identifies the instance to be disposed.

Return Value

- `DDS_RETCODE_OK`: The sample is written and the instance is marked for deletion.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: At least one of the arguments is invalid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_TIMEOUT`: Either the current action overflowed the available resources as specified by the combination of the reliability QoS policy, history QoS policy and resource_limits QoS policy, or the current action was waiting for data delivery acknowledgement by synchronous readers. This caused blocking of this operation, which could not be resolved before max_blocking_time of the reliability QoS policy elapsed.

dds_return_t **dds_dispose_ts** (*dds_entity_t writer*, **const** void **data*, *dds_time_t timestamp*)

This operation disposes an instance with a specific timestamp, identified by the data sample.

This operation performs the same functions as dds_dispose except that the application provides the value for the source_timestamp that is made available to connected reader objects. This timestamp is important for the interpretation of the destination_order QoS policy.

Return A dds_return_t indicating success or failure.

Parameters

- `writer`: The writer to dispose the data instance from.
- `data`: The data sample that identifies the instance to be disposed.
- `timestamp`: The timestamp used as source timestamp.

Return Value

- `DDS_RETCODE_OK`: The sample is written and the instance is marked for deletion
- `DDS_RETCODE_ERROR`: An internal error has occurred
- `DDS_RETCODE_BAD_PARAMETER`: At least one of the arguments is invalid
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted

- DDS_RETCODE_TIMEOUT: Either the current action overflowed the available resources as specified by the combination of the reliability QoS policy, history QoS policy and resource_limits QoS policy, or the current action was waiting for data delivery acknowledgment by synchronous readers. This caused blocking of this operation, which could not be resolved before max_blocking_time of the reliability QoS policy elapsed.

dds_return_t **dds_dispose_ih** (*dds_entity_t writer*, *dds_instance_handle_t handle*)

This operation disposes an instance, identified by the instance handle.

This operation requests the Data Distribution Service to modify the instance and mark it for deletion. Copies of the instance and its corresponding samples, which are stored in every connected reader and, dependent on the QoS policy settings (also in the Transient and Persistent stores) will be modified and marked for deletion by setting their dds_instance_state_t to DDS_IST_NOT_ALIVE_DISPOSED.

The given instance handle must correspond to the value that was returned by either the dds_register_instance operation, dds_register_instance_ts or dds_lookup_instance. If there is no correspondence, then the result of the operation is unspecified.

Return A dds_return_t indicating success or failure.

Parameters

- writer: The writer to dispose the data instance from.
- handle: The handle to identify an instance.

Return Value

- DDS_RETCODE_OK: The sample is written and the instance is marked for deletion.
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: At least one of the arguments is invalid
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted
- DDS_RETCODE_PRECONDITION_NOT_MET: The instance handle has not been registered with this writer

dds_return_t **dds_dispose_ih_ts** (*dds_entity_t writer*, *dds_instance_handle_t handle*, *dds_time_t timestamp*)

This operation disposes an instance with a specific timestamp, identified by the instance handle.

This operation performs the same functions as dds_dispose_ih except that the application provides the value for the source_timestamp that is made available to connected reader objects. This timestamp is important for the interpretation of the destination_order QoS policy.

Return A dds_return_t indicating success or failure.

Parameters

- writer: The writer to dispose the data instance from.
- handle: The handle to identify an instance.
- timestamp: The timestamp used as source timestamp.

Return Value

- DDS_RETCODE_OK: The sample is written and the instance is marked for deletion.
- DDS_RETCODE_ERROR: An internal error has occurred.

- DDS_RETCODE_BAD_PARAMETER: At least one of the arguments is invalid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.
- DDS_RETCODE_PRECONDITION_NOT_MET: The instance handle has not been registered with this writer.

dds_return_t **dds_write** (*dds_entity_t writer*, **const** void **data*)
Write the value of a data instance.

With this API, the value of the source timestamp is automatically made available to the data reader by the service.

Return dds_return_t indicating success or failure.

Parameters

- writer: The writer entity.
- data: Value to be written.

void **dds_write_flush** (*dds_entity_t writer*)

dds_return_t **dds_writecdr** (*dds_entity_t writer*, **struct** ddsi_serdata **serdata*)
Write a CDR serialized value of a data instance.

Return A dds_return_t indicating success or failure.

Parameters

- writer: The writer entity.
- cdr: CDR serialized value to be written.
- size: Size (in bytes) of CDR encoded data to be written.

dds_return_t **dds_write_ts** (*dds_entity_t writer*, **const** void **data*, *dds_time_t timestamp*)
Write the value of a data instance along with the source timestamp passed.

Return A dds_return_t indicating success or failure.

Parameters

- writer: The writer entity.
- data: Value to be written.
- timestamp: Source timestamp.

dds_entity_t **dds_create_readcondition** (*dds_entity_t reader*, uint32_t *mask*)
Creates a readcondition associated to the given reader.

The readcondition allows specifying which samples are of interest in a data reader's history, by means of a mask. The mask is or'd with the flags that are dds_sample_state_t, dds_view_state_t and dds_instance_state_t.

Based on the mask value set, the readcondition gets triggered when data is available on the reader.

Waitsets allow waiting for an event on some of any set of entities. This means that the readcondition can be used to wake up a waitset when data is in the reader history with states that matches the given mask.

Note The parent reader and every of its associated conditions (whether they are readconditions or queryconditions) share the same resources. This means that one of these entities reads or takes data, the states of the data will change for other entities automatically. For instance, if one reads a sample, then the sample state will become 'read' for all associated reader/conditions. Or if one takes a sample, then it's not available to any other associated reader/condition.

Return A valid condition handle or an error code.

Parameters

- `reader`: Reader to associate the condition to.
- `mask`: Interest (dds_sample_state_t|dds_view_state_t|dds_instance_state_t).

Return Value

- `>0`: A valid condition handle
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_create_querycondition** (*dds_entity_t reader*, uint32_t *mask*, *dds_querycondition_filter_fn filter*)

Creates a queryondition associated to the given reader.

The queryondition allows specifying which samples are of interest in a data reader's history, by means of a mask and a filter. The mask is or'd with the flags that are dds_sample_state_t, dds_view_state_t and dds_instance_state_t.

Based on the mask value set and data that matches the filter, the querycondition gets triggered when data is available on the reader.

Waitsets allow waiting for an event on some of any set of entities. This means that the querycondition can be used to wake up a waitset when data is in the reader history with states that matches the given mask and filter.

Note The parent reader and every of its associated conditions (whether they are readconditions or queryconditions) share the same resources. This means that one of these entities reads or takes data, the states of the data will change for other entities automatically. For instance, if one reads a sample, then the sample state will become 'read' for all associated reader/conditions. Or if one takes a sample, then it's not available to any other associated reader/condition.

Return A valid condition handle or an error code

Parameters

- `reader`: Reader to associate the condition to.
- `mask`: Interest (dds_sample_state_t|dds_view_state_t|dds_instance_state_t).
- `filter`: Callback that the application can use to filter specific samples.

Return Value

- `>=0`: A valid condition handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_create_guardcondition** (*dds_entity_t participant*)
Creates a guardcondition.

Waitsets allow waiting for an event on some of any set of entities. This means that the guardcondition can be used to wake up a waitset when data is in the reader history with states that matches the given mask.

Return A valid condition handle or an error code.

Return Value

- `>0`: A valid condition handle
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_set_guardcondition** (*dds_entity_t guardcond*, bool *triggered*)
Sets the trigger status of a guardcondition.

Return Value

- `DDS_RETCODE_OK`: Operation successful
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_read_guardcondition** (*dds_entity_t guardcond*, bool **triggered*)
Reads the trigger status of a guardcondition.

Return Value

- `DDS_RETCODE_OK`: Operation successful
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_take_guardcondition** (*dds_entity_t guardcond*, bool **triggered*)
Reads and resets the trigger status of a guardcondition.

Return Value

- `DDS_RETCODE_OK`: Operation successful
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_entity_t **dds_create_waitset** (*dds_entity_t participant*)
Create a waitset and allocate the resources required.

A WaitSet object allows an application to wait until one or more of the conditions of the attached entities evaluates to TRUE or until the timeout expires.

Return A valid waitset handle or an error code.

Parameters

- `participant`: Domain participant which the WaitSet contains.

Return Value

- `>=0`: A valid waitset handle.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_waitset_get_entities** (*dds_entity_t waitset*, *dds_entity_t *entities*, size_t *size*)

Acquire previously attached entities.

This functions takes a pre-allocated list to put the entities in and will return the number of found entities. It is possible that the given size of the list is not the same as the number of found entities. If less entities are found, then the last few entries in the list are untouched. When more entities are found, then only 'size' number of entries are inserted into the list, but still the complete count of the found entities is returned. Which entities are returned in the latter case is undefined.

Return A dds_return_t with the number of children or an error code.

Parameters

- `waitset`: Waitset from which to get its attached entities.
- `entities`: Pre-allocated array to contain the found entities.
- `size`: Size of the pre-allocated entities' list.

Return Value

- `>=0`: Number of children found (can be larger than 'size').
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The entities parameter is NULL, while a size is provided.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The waitset has already been deleted.

dds_return_t **dds_waitset_attach** (*dds_entity_t waitset*, *dds_entity_t entity*, *dds_attach_t x*)

This operation attaches an Entity to the WaitSet.

This operation attaches an Entity to the WaitSet. The *dds_waitset_wait()* will block when none of the attached entities are triggered. 'Triggered' (*dds_triggered()*) doesn't mean the same for every entity:

- Reader/Writer/Publisher/Subscriber/Topic/Participant
  - These are triggered when their status changed.
- WaitSet
  - Triggered when trigger value was set to true by the application. It stays triggered until application sets the trigger value to false (*dds_waitset_set_trigger()*). This can be used to wake up an waitset for different reasons (f.i. termination) than the 'normal' status change (like new data).
- ReadCondition/QueryCondition
  - Triggered when data is available on the related Reader that matches the Condition.

Multiple entities can be attached to a single waitset. A particular entity can be attached to multiple waitsets. However, a particular entity can not be attached to a particular waitset multiple times.

Return A dds_return_t indicating success or failure.

Parameters

- `waitset`: The waitset to attach the given entity to.
- `entity`: The entity to attach.
- `x`: Blob that will be supplied when the waitset wait is triggerd by the given entity.

Return Value

- `DDS_RETCODE_OK`: Entity attached.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The given waitset or entity are not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The waitset has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The entity was already attached.

dds_return_t **dds_waitset_detach** (*dds_entity_t waitset*, *dds_entity_t entity*)
This operation detaches an Entity to the WaitSet.

Return A dds_return_t indicating success or failure.

Parameters

- `waitset`: The waitset to detach the given entity from.
- `entity`: The entity to detach.

Return Value

- `DDS_RETCODE_OK`: Entity attached.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The given waitset or entity are not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The waitset has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The entity is not attached.

dds_return_t **dds_waitset_set_trigger** (*dds_entity_t waitset*, bool *trigger*)
Sets the trigger_value associated with a waitset.

When the waitset is attached to itself and the trigger value is set to 'true', then the waitset will wake up just like with an other status change of the attached entities.

This can be used to forcefully wake up a waitset, for instance when the application wants to shut down. So, when the trigger value is true, the waitset will wake up or not wait at all.

The trigger value will remain true until the application sets it false again deliberately.

Return A dds_return_t indicating success or failure.

Parameters

- `waitset`: The waitset to set the trigger value on.
- `trigger`: The trigger value to set.

Return Value

- `DDS_RETCODE_OK`: Entity attached.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The given waitset is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The waitset has already been deleted.

dds_return_t **dds_waitset_wait** (*dds_entity_t waitset*, *dds_attach_t *xs*, size_t *nxs*, *dds_duration_t reltimeout*)

This operation allows an application thread to wait for the a status change or other trigger on (one of) the entities that are attached to the WaitSet.

The "dds_waitset_wait" operation blocks until the some of the attached entities have triggered or "reltimeout" has elapsed. 'Triggered' (*dds_triggered()*) doesn't mean the same for every entity:

- Reader/Writer/Publisher/Subscriber/Topic/Participant
  - These are triggered when their status changed.
- WaitSet
  - Triggered when trigger value was set to true by the application. It stays triggered until application sets the trigger value to false (*dds_waitset_set_trigger()*). This can be used to wake up an waitset for different reasons (f.i. termination) than the 'normal' status change (like new data).
- ReadCondition/QueryCondition
  - Triggered when data is available on the related Reader that matches the Condition.

This functions takes a pre-allocated list to put the "xs" blobs in (that were provided during the attach of the related entities) and will return the number of triggered entities. It is possible that the given size of the list is not the same as the number of triggered entities. If less entities were triggered, then the last few entries in the list are untouched. When more entities are triggered, then only 'size' number of entries are inserted into the list, but still the complete count of the triggered entities is returned. Which "xs" blobs are returned in the latter case is undefined.

In case of a time out, the return value is 0.

Deleting the waitset while the application is blocked results in an error code (i.e. < 0) returned by "wait".

Multiple threads may block on a single waitset at the same time; the calls are entirely independent.

An empty waitset never triggers (i.e., dds_waitset_wait on an empty waitset is essentially equivalent to a sleep).

The "dds_waitset_wait_until" operation is the same as the "dds_waitset_wait" except that it takes an absolute timeout.

Return A dds_return_t with the number of entities triggered or an error code

Parameters

- `waitset`: The waitset to set the trigger value on.
- `xs`: Pre-allocated list to store the 'blobs' that were provided during the attach of the triggered entities.

- `nxs`: The size of the pre-allocated blobs list.
- `reltimeout`: Relative timeout

Return Value

- `>0`: Number of entities triggered.
- `0`: Time out (no entities were triggered).
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The given waitset is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The waitset has already been deleted.

dds_return_t **dds_waitset_wait_until** (*dds_entity_t waitset*, *dds_attach_t *xs*, size_t *nxs*, *dds_time_t abstimeout*)

This operation allows an application thread to wait for the a status change or other trigger on (one of) the entities that are attached to the WaitSet.

The "dds_waitset_wait" operation blocks until the some of the attached entities have triggered or "abstimeout" has been reached. 'Triggered' (*dds_triggered()*) doesn't mean the same for every entity:

- Reader/Writer/Publisher/Subscriber/Topic/Participant
  - These are triggered when their status changed.
- WaitSet
  - Triggered when trigger value was set to true by the application. It stays triggered until application sets the trigger value to false (*dds_waitset_set_trigger()*). This can be used to wake up an waitset for different reasons (f.i. termination) than the 'normal' status change (like new data).
- ReadCondition/QueryCondition
  - Triggered when data is available on the related Reader that matches the Condition.

This functions takes a pre-allocated list to put the "xs" blobs in (that were provided during the attach of the related entities) and will return the number of triggered entities. It is possible that the given size of the list is not the same as the number of triggered entities. If less entities were triggered, then the last few entries in the list are untouched. When more entities are triggered, then only 'size' number of entries are inserted into the list, but still the complete count of the triggered entities is returned. Which "xs" blobs are returned in the latter case is undefined.

In case of a time out, the return value is 0.

Deleting the waitset while the application is blocked results in an error code (i.e. < 0) returned by "wait".

Multiple threads may block on a single waitset at the same time; the calls are entirely independent.

An empty waitset never triggers (i.e., dds_waitset_wait on an empty waitset is essentially equivalent to a sleep).

The "dds_waitset_wait" operation is the same as the "dds_waitset_wait_until" except that it takes an relative timeout.

The "dds_waitset_wait" operation is the same as the "dds_wait" except that it takes an absolute timeout.

Return A dds_return_t with the number of entities triggered or an error code.

Parameters

- `waitset`: The waitset to set the trigger value on.

- `xs`: Pre-allocated list to store the 'blobs' that were provided during the attach of the triggered entities.
- `nxs`: The size of the pre-allocated blobs list.
- `abstimeout`: Absolute timeout

Return Value

- `>0`: Number of entities triggered.
- `0`: Time out (no entities were triggered).
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: The given waitset is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The waitset has already been deleted.

dds_return_t **dds_read** (*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, size_t *bufsz*, uint32_t *maxs*)

Access and read the collection of data values (of same type) and sample info from the data reader, readcondition or querycondition.

Return value provides information about number of samples read, which will be <= maxs. Based on the count, the buffer will contain data to be read only when valid_data bit in sample info structure is set. The buffer required for data values, could be allocated explicitly or can use the memory from data reader to prevent copy. In the latter case, buffer and sample_info should be returned back, once it is no longer using the Data. Data values once read will remain in the buffer with the sample_state set to READ and view_state set to NOT_NEW.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_read_wl** (*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, uint32_t *maxs*)

Access and read loaned samples of data reader, readcondition or querycondition.

After dds_read_wl function is being called and the data has been handled, dds_return_loan function must be called to possibly free memory.

Return A dds_return_t with the number of samples read or an error code

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL)
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value
- `maxs`: Maximum number of samples to read

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_read_mask** (*dds_entity_t reader_or_condition*, void ****buf*, *dds_sample_info_t *si*, size_t *bufsz*, uint32_t *maxs*, uint32_t *mask*)

Read the collection of data values and sample info from the data reader, readcondition or querycondition based on mask.

When using a readcondition or querycondition, their masks are or'd with the given mask.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.
- `mask`: Filter the data based on dds_sample_state_tldds_view_state_tldds_instance_state_t.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_read_mask_wl** (*dds_entity_t reader_or_condition*, void ****buf*, *dds_sample_info_t *si*, uint32_t *maxs*, uint32_t *mask*)

Access and read loaned samples of data reader, readcondition or querycondition based on mask.

When using a readcondition or querycondition, their masks are or'd with the given mask.

After dds_read_mask_wl function is being called and the data has been handled, dds_return_loan function must be called to possibly free memory

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `maxs`: Maximum number of samples to read.
- `mask`: Filter the data based on dds_sample_state_tldds_view_state_tldds_instance_state_t.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_read_instance** (*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, size_t *bufsz*, uint32_t *maxs*, *dds_instance_handle_t handle*)

Access and read the collection of data values (of same type) and sample info from the data reader, readcondition or querycondition, coped by the provided instance handle.

This operation implements the same functionality as dds_read, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.
- `handle`: Instance handle related to the samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The instance handle has not been registered with this reader.

dds_return_t **dds_read_instance_wl**(*dds_entity_t reader_or_condition*, void **buf, *dds_sample_info_t* *si, uint32_t *maxs*, *dds_instance_handle_t handle*)

Access and read loaned samples of data reader, readcondition or querycondition, scoped by the provided instance handle.

This operation implements the same functionality as dds_read_wl, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `maxs`: Maximum number of samples to read.
- `handle`: Instance handle related to the samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The instance handle has not been registered with this reader.

dds_return_t **dds_read_instance_mask**(*dds_entity_t reader_or_condition*, void **buf, *dds_sample_info_t* *si, size_t *bufsz*, uint32_t *maxs*, *dds_instance_handle_t handle*, uint32_t *mask*)

Read the collection of data values and sample info from the data reader, readcondition or querycondition based on mask and scoped by the provided instance handle.

This operation implements the same functionality as dds_read_mask, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.
- `handle`: Instance handle related to the samples to read.
- `mask`: Filter the data based on dds_sample_state_tldds_view_state_tldds_instance_state_t.

Return Value

- >=0: Number of samples read.
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: One of the given arguments is not valid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.
- DDS_RETCODE_PRECONDITION_NOT_MET: The instance handle has not been registered with this reader.

dds_return_t **dds_read_instance_mask_wl**(*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, uint32_t *maxs*, *dds_instance_handle_t handle*, uint32_t *mask*)

Access and read loaned samples of data reader, readcondition or querycondition based on mask, scoped by the provided instance handle.

This operation implements the same functionality as dds_read_mask_wl, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- reader_or_condition: Reader, readcondition or querycondition entity.
- buf: An array of pointers to samples into which data is read (pointers can be NULL).
- si: Pointer to an array of *dds_sample_info_t* returned for each data value.
- maxs: Maximum number of samples to read.
- handle: Instance handle related to the samples to read.
- mask: Filter the data based on dds_sample_state_tldds_view_state_tldds_instance_state_t.

Return Value

- >=0: Number of samples read.
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: One of the given arguments is not valid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.
- DDS_RETCODE_PRECONDITION_NOT_MET: The instance handle has not been registered with this reader.

dds_return_t **dds_take**(*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, size_t *bufsz*, uint32_t *maxs*)

Access the collection of data values (of same type) and sample info from the data reader, readcondition or querycondition.

Data value once read is removed from the Data Reader cannot to 'read' or 'taken' again. Return value provides information about number of samples read, which will be <= maxs. Based on the count, the buffer will contain data to be read only when valid_data bit in sample info structure is set. The buffer required for data values, could be allocated explicitly or can use the memory from data reader to prevent copy. In the latter case, buffer and sample_info should be returned back, once it is no longer using the Data.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_take_wl**(*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, uint32_t *maxs*)

Access loaned samples of data reader, readcondition or querycondition.

After dds_take_wl function is being called and the data has been handled, dds_return_loan function must be called to possibly free memory

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `maxs`: Maximum number of samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_take_mask**(*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, size_t *bufsz*, uint32_t *maxs*, uint32_t *mask*)

Take the collection of data values (of same type) and sample info from the data reader, readcondition or querycondition based on mask.

When using a readcondition or querycondition, their masks are or'd with the given mask.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.
- `mask`: Filter the data based on dds_sample_state_t|dds_view_state_t|dds_instance_state_t.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_take_mask_wl** (*dds_entity_t reader_or_condition*, void ***buf*, *dds_sample_info_t* **si*, uint32_t *maxs*, uint32_t *mask*)

Access loaned samples of data reader, readcondition or querycondition based on mask.

When using a readcondition or querycondition, their masks are or'd with the given mask.

After dds_take_mask_wl function is being called and the data has been handled, dds_return_loan function must be called to possibly free memory

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `maxs`: Maximum number of samples to read.
- `mask`: Filter the data based on dds_sample_state_t|dds_view_state_t|dds_instance_state_t.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

int **dds_takecdr** (*dds_entity_t reader_or_condition*, **struct** ddsi_serdata ***buf*, uint32_t *maxs*, *dds_sample_info_t* **si*, uint32_t *mask*)

dds_return_t **dds_take_instance** (*dds_entity_t reader_or_condition*, void ***buf*, *dds_sample_info_t* **si*, size_t *bufsz*, uint32_t *maxs*, *dds_instance_handle_t handle*)

Access the collection of data values (of same type) and sample info from the data reader, readcondition or querycondition but scoped by the given instance handle.

This operation mplements the same functionality as dds_take, except that only data scoped to the provided instance handle is taken.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `bufsz`: The size of buffer provided.
- `maxs`: Maximum number of samples to read.
- `handle`: Instance handle related to the samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The instance handle has not been registered with this reader.

dds_return_t **dds_take_instance_wl**(*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, uint32_t *maxs*, *dds_instance_handle_t handle*)

Access loaned samples of data reader, readcondition or querycondition, scoped by the given instance handle.

This operation implements the same functionality as dds_take_wl, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- `reader_or_condition`: Reader, readcondition or querycondition entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: Pointer to an array of *dds_sample_info_t* returned for each data value.
- `maxs`: Maximum number of samples to read.
- `handle`: Instance handle related to the samples to read.

Return Value

- `>=0`: Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

- DDS_RETCODE_PRECONDITION_NOT_MET: The instance handle has not been registered with this reader.

dds_return_t **dds_take_instance_mask** (*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, size_t *bufsz*, uint32_t *maxs*, *dds_instance_handle_t handle*, uint32_t *mask*)

Take the collection of data values (of same type) and sample info from the data reader, readcondition or querycondition based on mask and scoped by the given instance handle.

This operation implements the same functionality as dds_take_mask, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples read or an error code.

Parameters

- reader_or_condition: Reader, readcondition or querycondition entity.
- buf: An array of pointers to samples into which data is read (pointers can be NULL).
- si: Pointer to an array of *dds_sample_info_t* returned for each data value.
- bufsz: The size of buffer provided.
- maxs: Maximum number of samples to read.
- handle: Instance handle related to the samples to read.
- mask: Filter the data based on dds_sample_state_t|dds_view_state_t|dds_instance_state_t.

Return Value

- >=0: Number of samples read.
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: One of the given arguments is not valid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.
- DDS_RETCODE_PRECONDITION_NOT_MET: The instance handle has not been registered with this reader.

dds_return_t **dds_take_instance_mask_wl** (*dds_entity_t reader_or_condition*, void **buf*, *dds_sample_info_t *si*, uint32_t *maxs*, *dds_instance_handle_t handle*, uint32_t *mask*)

Access loaned samples of data reader, readcondition or querycondition based on mask and scoped by the given intance handle.

This operation implements the same functionality as dds_take_mask_wl, except that only data scoped to the provided instance handle is read.

Return A dds_return_t with the number of samples or an error code.

Parameters

- reader_or_condition: Reader, readcondition or querycondition entity.
- buf: An array of pointers to samples into which data is read (pointers can be NULL).
- si: Pointer to an array of *dds_sample_info_t* returned for each data value.
- maxs: Maximum number of samples to read.

- `handle`: Instance handle related to the samples to read.
- `mask`: Filter the data based on dds_sample_state_tldds_view_state_tldds_instance_state_t.

Return Value

- `>=`: 0 Number of samples read.
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.
- `DDS_RETCODE_PRECONDITION_NOT_MET`: The instance handle has not been registered with this reader.

dds_return_t **dds_take_next** (*dds_entity_t reader*, void **buf*, *dds_sample_info_t *si*)

Read, copy and remove the status set for the entity.

This operation copies the next, non-previously accessed data value and corresponding sample info and removes from the data reader. As an entity, only reader is accepted.

Return A dds_return_t indicating success or failure.

Parameters

- `reader`: The reader entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: The pointer to *dds_sample_info_t* returned for a data value.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_take_next_wl** (*dds_entity_t reader*, void **buf*, *dds_sample_info_t *si*)

Read, copy and remove the status set for the entity.

This operation copies the next, non-previously accessed data value and corresponding sample info and removes from the data reader. As an entity, only reader is accepted.

After dds_take_next_wl function is being called and the data has been handled, dds_return_loan function must be called to possibly free memory.

Return A dds_return_t indicating success or failure.

Parameters

- `reader`: The reader entity.
- `buf`: An array of pointers to samples into which data is read (pointers can be NULL).
- `si`: The pointer to *dds_sample_info_t* returned for a data value.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.

- DDS_RETCODE_BAD_PARAMETER: The entity parameter is not a valid parameter.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_return_t **dds_read_next** (*dds_entity_t reader*, void **buf*, *dds_sample_info_t *si*)

Read and copy the status set for the entity.

This operation copies the next, non-previously accessed data value and corresponding sample info. As an entity, only reader is accepted.

Return A dds_return_t indicating success or failure.

Parameters

- reader: The reader entity.
- buf: An array of pointers to samples into which data is read (pointers can be NULL).
- si: The pointer to *dds_sample_info_t* returned for a data value.

Return Value

- DDS_RETCODE_OK: The operation was successful.
- DDS_RETCODE_BAD_PARAMETER: The entity parameter is not a valid parameter.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_return_t **dds_read_next_wl** (*dds_entity_t reader*, void **buf*, *dds_sample_info_t *si*)

Read and copy the status set for the loaned sample.

This operation copies the next, non-previously accessed data value and corresponding loaned sample info. As an entity, only reader is accepted.

After dds_read_next_wl function is being called and the data has been handled, dds_return_loan function must be called to possibly free memory.

Return A dds_return_t indicating success or failure.

Parameters

- reader: The reader entity.
- buf: An array of pointers to samples into which data is read (pointers can be NULL).
- si: The pointer to *dds_sample_info_t* returned for a data value.

Return Value

- DDS_RETCODE_OK: The operation was successful.
- DDS_RETCODE_BAD_PARAMETER: The entity parameter is not a valid parameter.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_return_t **dds_return_loan** (*dds_entity_t reader_or_condition*, void **buf*, int32_t *bufsz*)

Return loaned samples to data-reader or condition associated with a data-reader.

Used to release sample buffers returned by a read/take operation. When the application provides an empty buffer, memory is allocated and managed by DDS. By calling dds_return_loan, the memory is released so

that the buffer can be reused during a successive read/take operation. When a condition is provided, the reader to which the condition belongs is looked up.

Return A dds_return_t indicating success or failure

Parameters

- `rd_or_cnd`: Reader or condition that belongs to a reader.
- `buf`: An array of (pointers to) samples.
- `bufsz`: The number of (pointers to) samples stored in buf.

dds_instance_handle_t **dds_lookup_instance** (*dds_entity_t entity*, **const** void **data*)
This operation takes a sample and returns an instance handle to be used for subsequent operations.

Return instance handle or DDS_HANDLE_NIL if instance could not be found from key.

Parameters

- `entity`: Reader or Writer entity.
- `data`: Sample with a key fields set.

dds_instance_handle_t **dds_instance_lookup** (*dds_entity_t entity*, **const** void **data*)

dds_return_t **dds_instance_get_key** (*dds_entity_t entity*, *dds_instance_handle_t inst*, void **data*)
This operation takes an instance handle and return a key-value corresponding to it.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: Reader or writer entity.
- `inst`: Instance handle.
- `data`: pointer to an instance, to which the key ID corresponding to the instance handle will be returned, the sample in the instance should be ignored.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: One of the parameters was invalid or the topic does not exist.
- `DDS_RETCODE_ERROR`: An internal error has occurred.

dds_return_t **dds_begin_coherent** (*dds_entity_t entity*)
Begin coherent publishing or begin accessing a coherent set in a subscriber.

Invoking on a Writer or Reader behaves as if dds_begin_coherent was invoked on its parent Publisher or Subscriber respectively.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: The entity that is prepared for coherent access.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.

- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: The provided entity is invalid or not supported.

dds_return_t **dds_end_coherent** (*dds_entity_t entity*)
End coherent publishing or end accessing a coherent set in a subscriber.

Invoking on a Writer or Reader behaves as if dds_end_coherent was invoked on its parent Publisher or Subscriber respectively.

Return A dds_return_t indicating success or failure.

Parameters

- entity: The entity on which coherent access is finished.

Return Value

- DDS_RETCODE_OK: The operation was successful.
- DDS_RETCODE_BAD_PARAMETER: The provided entity is invalid or not supported.

dds_return_t **dds_notify_readers** (*dds_entity_t subscriber*)
Trigger DATA_AVAILABLE event on contained readers.

The DATA_AVAILABLE event is broadcast to all readers owned by this subscriber that currently have new data available. Any on_data_available listener callbacks attached to respective readers are invoked.

Return A dds_return_t indicating success or failure.

Parameters

- subscriber: A valid subscriber handle.

Return Value

- DDS_RETCODE_OK: The operation was successful.
- DDS_RETCODE_BAD_PARAMETER: The provided subscriber is invalid.

dds_return_t **dds_triggered** (*dds_entity_t entity*)
Checks whether the entity has one of its enabled statuses triggered.

Return A dds_return_t indicating success or failure.

Parameters

- entity: Entity for which to check for triggered status.

Return Value

- DDS_RETCODE_OK: The operation was successful.
- DDS_RETCODE_BAD_PARAMETER: The entity parameter is not a valid parameter.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_entity_t **dds_get_topic** (*dds_entity_t entity*)
Get the topic.

This operation returns a topic (handle) when the function call is done with reader, writer, read condition or query condition. For instance, it will return the topic when it is used for creating the reader or writer.

For the conditions, it returns the topic that is used for creating the reader which was used to create the condition.

Return A dds_return_t indicating success or failure.

Parameters

- `entity`: The entity.

Return Value

- `DDS_RETCODE_OK`: The operation was successful.
- `DDS_RETCODE_BAD_PARAMETER`: The entity parameter is not a valid parameter.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

Variables

const *dds_entity_t* **DDS_BUILTIN_TOPIC_DCPSPARTICIPANT**

const *dds_entity_t* **DDS_BUILTIN_TOPIC_DCPSTOPIC**

const *dds_entity_t* **DDS_BUILTIN_TOPIC_DCPSPUBLICATION**

const *dds_entity_t* **DDS_BUILTIN_TOPIC_DCPSSUBSCRIPTION**

file **dds_public_alloc.h**

#include "os/os_public.h"#include "ddsc/dds_export.h" DDS C Allocation API.

This header file defines the public API of allocation convenience functions in the Eclipse Cyclone DDS C language binding.

Defines

DDS_FREE_KEY_BIT

DDS_FREE_CONTENTS_BIT

DDS_FREE_ALL_BIT

Typedefs

typedef struct *dds_allocator* **dds_allocator_t**

typedef struct *dds_aligned_allocator* **dds_aligned_allocator_t**

typedef void *(***dds_alloc_fn_t**) (size_t)

typedef void *(***dds_realloc_fn_t**) (void *, size_t)

typedef void (***dds_free_fn_t**) (void *)

Enums

enum dds_free_op_t

Values:

DDS_FREE_ALL = DDS_FREE_KEY_BIT | DDS_FREE_CONTENTS_BIT | DDS_FREE_ALL_BIT

DDS_FREE_CONTENTS = DDS_FREE_KEY_BIT | DDS_FREE_CONTENTS_BIT

DDS_FREE_KEY = DDS_FREE_KEY_BIT

Functions

void **dds_set_allocator**(**const** *dds_allocator_t* *n, *dds_allocator_t* *o)

void **dds_set_aligned_allocator**(**const** *dds_aligned_allocator_t* *n, *dds_aligned_allocator_t* *o)

void ***dds_alloc**(size_t *size*)

void ***dds_realloc**(void *ptr, size_t *size*)

void ***dds_realloc_zero**(void *ptr, size_t *size*)

void **dds_free**(void *ptr)

char ***dds_string_alloc**(size_t *size*)

char ***dds_string_dup**(**const** char *str)

void **dds_string_free**(char *str)

void **dds_sample_free**(void *sample, **const struct** *dds_topic_descriptor* *desc, *dds_free_op_t* op)

file **dds_public_error.h**

#include "os/os_public.h"#include "ddsc/dds_export.h" DDS C Error API.

This header file defines the public API of error values and convenience functions in the CycloneDDS C language binding.

Return codes

DDS_RETCODE_OK
Success

DDS_RETCODE_ERROR
Non specific error

DDS_RETCODE_UNSUPPORTED
Feature unsupported

DDS_RETCODE_BAD_PARAMETER
Bad parameter value

DDS_RETCODE_PRECONDITION_NOT_MET
Precondition for operation not met

DDS_RETCODE_OUT_OF_RESOURCES
When an operation fails because of a lack of resources

DDS_RETCODE_NOT_ENABLED
When a configurable feature is not enabled

DDS_RETCODE_IMMUTABLE_POLICY
When an attempt is made to modify an immutable policy

DDS_RETCODE_INCONSISTENT_POLICY
When a policy is used with inconsistent values

DDS_RETCODE_ALREADY_DELETED
When an attempt is made to delete something more than once

DDS_RETCODE_TIMEOUT
When a timeout has occurred

DDS_RETCODE_NO_DATA
When expected data is not provided

DDS_RETCODE_ILLEGAL_OPERATION
When a function is called when it should not be

DDS_RETCODE_NOT_ALLOWED_BY_SECURITY
When credentials are not enough to use the function

DDS_Error_Type

DDS_CHECK_REPORT

DDS_CHECK_FAIL

DDS_CHECK_EXIT

Macros for error handling

DDS_TO_STRING(n)

DDS_INT_TO_STRING(n)

Defines

DDS_ERR_NR_MASK

DDS_ERR_LINE_MASK

DDS_ERR_FILE_ID_MASK

DDS_SUCCESS

dds_err_nr(e)
Macro to extract error number

dds_err_line(e)
Macro to extract line number

dds_err_file_id(e)
Macro to extract file identifier

DDS_ERR_CHECK(e, f)
Macro that defines dds_err_check function

DDS_FAIL(m)
Macro that defines dds_fail function

Typedefs

typedef void (***dds_fail_fn**)(**const** char *, **const** char *)
Failure handler

Functions

const char ***dds_err_str**(*dds_return_t err*)
Takes the error value and outputs a string corresponding to it.

Return String corresponding to the error value

Parameters

- `err`: Error value to be converted to a string

bool **dds_err_check**(*dds_return_t err*, unsigned *flags*, **const** char **where*)
Takes the error number, error type and filename and line number and formats it to a string which can be used for debugging.

Return true - True

Return false - False

Parameters

- `err`: Error value
- `flags`: Indicates Fail, Exit or Report
- `where`: File and line number

void **dds_fail_set**(*dds_fail_fn fn*)
Set the failure function.

Parameters

- `fn`: Function to invoke on failure

dds_fail_fn **dds_fail_get**(void)
Get the failure function.

Return Failure function

void **dds_fail**(**const** char **msg*, **const** char **where*)
Handles failure through an installed failure handler.

[in] msg String containing failure message [in] where String containing file and location

file **dds_public_impl.h**
#include "ddsc/dds_public_alloc.h"#include "ddsc/dds_public_stream.h"#include "os/os_public.h"#include "ddsc/dds_export.h" DDS C Implementation API.

This header file defines the public API for all kinds of things in the Eclipse Cyclone DDS C language binding.

Defines

DDS_LENGTH_UNLIMITED

DDS_TOPIC_NO_OPTIMIZE

DDS_TOPIC_FIXED_KEY

DDS_READ_SAMPLE_STATE

DDS_NOT_READ_SAMPLE_STATE

DDS_ANY_SAMPLE_STATE

DDS_NEW_VIEW_STATE

DDS_NOT_NEW_VIEW_STATE

DDS_ANY_VIEW_STATE

DDS_ALIVE_INSTANCE_STATE

DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE

DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE

DDS_ANY_INSTANCE_STATE

DDS_ANY_STATE

DDS_DOMAIN_DEFAULT

DDS_HANDLE_NIL

DDS_ENTITY_NIL

DDS_ENTITY_KIND_MASK

DDS_OP_RTS

DDS_OP_ADR

DDS_OP_JSR

DDS_OP_JEQ

DDS_OP_VAL_1BY

DDS_OP_VAL_2BY

DDS_OP_VAL_4BY

DDS_OP_VAL_8BY

DDS_OP_VAL_STR

DDS_OP_VAL_BST

DDS_OP_VAL_SEQ

DDS_OP_VAL_ARR

DDS_OP_VAL_UNI

DDS_OP_VAL_STU

DDS_OP_TYPE_1BY

DDS_OP_TYPE_2BY

DDS_OP_TYPE_4BY

DDS_OP_TYPE_8BY

DDS_OP_TYPE_STR

DDS_OP_TYPE_SEQ

DDS_OP_TYPE_ARR

DDS_OP_TYPE_UNI

DDS_OP_TYPE_STU

DDS_OP_TYPE_BST

DDS_OP_TYPE_BOO

DDS_OP_SUBTYPE_BOO

DDS_OP_SUBTYPE_1BY

DDS_OP_SUBTYPE_2BY

DDS_OP_SUBTYPE_4BY

DDS_OP_SUBTYPE_8BY

DDS_OP_SUBTYPE_STR

DDS_OP_SUBTYPE_SEQ

DDS_OP_SUBTYPE_ARR

DDS_OP_SUBTYPE_UNI

DDS_OP_SUBTYPE_STU

DDS_OP_SUBTYPE_BST

DDS_OP_FLAG_KEY

DDS_OP_FLAG_DEF

Typedefs

typedef struct *dds_sequence* **dds_sequence_t**

typedef struct *dds_key_descriptor* **dds_key_descriptor_t**

typedef struct *dds_topic_descriptor* **dds_topic_descriptor_t**

typedef enum *dds_entity_kind* **dds_entity_kind_t**

typedef uint64_t **dds_instance_handle_t**

typedef int32_t **dds_domainid_t**

Enums

enum dds_entity_kind

Values:

DDS_KIND_DONTCARE = 0x00000000

DDS_KIND_TOPIC = 0x01000000

DDS_KIND_PARTICIPANT = 0x02000000

DDS_KIND_READER = 0x03000000

DDS_KIND_WRITER = 0x04000000

DDS_KIND_SUBSCRIBER = 0x05000000

DDS_KIND_PUBLISHER = 0x06000000

DDS_KIND_COND_READ = 0x07000000

DDS_KIND_COND_QUERY = 0x08000000

DDS_KIND_COND_GUARD = 0x09000000

DDS_KIND_WAITSET = 0x0A000000

DDS_KIND_INTERNAL = 0x0B000000

Functions

void **dds_write_set_batch**(bool *enable*)
Description : Enable or disable write batching. Overrides default configuration setting for write batching (DDSI2E/Internal/WriteBatch).

Arguments :

1. enable Enables or disables write batching for all writers.

void **dds_ssl_plugin**(void)
Description : Install tcp/ssl and encryption support. Depends on openssl.

Arguments :

1. None

void **dds_durability_plugin**(void)
Description : Install client durability support. Depends on OSPL server.

Arguments :

1. None

file **dds_public_listener.h**
#include "ddsc/dds_export.h"#include "ddsc/dds_public_impl.h"#include "ddsc/dds_public_status.h"#include "os/os_public.h" DDS C Listener API.

This header file defines the public API of listeners in the Eclipse Cyclone DDS C language binding.

Defines

DDS_LUNSET

Typedefs

typedef void (***dds_on_inconsistent_topic_fn**)(*dds_entity_t* topic, **const** *dds_inconsistent_topic_status_t* status, void *arg)

typedef void (***dds_on_liveliness_lost_fn**)(*dds_entity_t* writer, **const** *dds_liveliness_lost_status_t* status, void *arg)

typedef void (***dds_on_offered_deadline_missed_fn**) (*dds_entity_t* writer, **const** *dds_offered_deadline_missed_status_t* status, void *arg)

typedef void (***dds_on_offered_incompatible_qos_fn**) (*dds_entity_t* writer, **const** *dds_offered_incompatible_qos_status_t* status, void *arg)

typedef void (***dds_on_data_on_readers_fn**) (*dds_entity_t* subscriber, void *arg)

typedef void (***dds_on_sample_lost_fn**) (*dds_entity_t* reader, **const** *dds_sample_lost_status_t* status, void *arg)

typedef void (***dds_on_data_available_fn**) (*dds_entity_t* reader, void *arg)

typedef void (***dds_on_sample_rejected_fn**) (*dds_entity_t* reader, **const** *dds_sample_rejected_status_t* status, void *arg)

typedef void (***dds_on_liveliness_changed_fn**) (*dds_entity_t* reader, **const** *dds_liveliness_changed_status_t* status, void *arg)

typedef void (***dds_on_requested_deadline_missed_fn**) (*dds_entity_t* reader, **const** *dds_requested_deadline_missed_status_t* status, void *arg)

typedef void (***dds_on_requested_incompatible_qos_fn**) (*dds_entity_t* reader, **const** *dds_requested_incompatible_qos_status_t* status, void *arg)

typedef void (***dds_on_publication_matched_fn**) (*dds_entity_t* writer, **const** *dds_publication_matched_status_t* status, void *arg)

typedef void (***dds_on_subscription_matched_fn**) (*dds_entity_t* reader, **const** *dds_subscription_matched_status_t* status, void *arg)

typedef struct dds_listener **dds_listener_t**

Functions

dds_listener_t ***dds_create_listener** (void **arg*)

Allocate memory and initializes to default values (::DDS_LUNSET) of a listener.

Return Returns a pointer to the allocated memory for dds_listener_t structure.

Parameters

- `arg`: optional pointer that will be passed on to the listener callbacks

dds_listener_t ***dds_listener_create** (void **arg*)

void **dds_delete_listener** (*dds_listener_t* **listener*)

Delete the memory allocated to listener structure.

Parameters

- `listener`: pointer to the listener struct to delete

void **dds_listener_delete** (*dds_listener_t* **listener*)

void **dds_reset_listener**(*dds_listener_t *listener*)
Reset the listener structure contents to ::DDS_LUNSET.

Parameters

- `listener`: pointer to the listener struct to reset

void **dds_listener_reset**(*dds_listener_t *listener*)

void **dds_copy_listener**(*dds_listener_t *dst*, **const** *dds_listener_t *src*)
Copy the listener callbacks from source to destination.

Parameters

- `dst`: The pointer to the destination listener structure, where the content is to copied
- `src`: The pointer to the source listener structure to be copied

void **dds_listener_copy**(*dds_listener_t *dst*, **const** *dds_listener_t *src*)

void **dds_merge_listener**(*dds_listener_t *dst*, **const** *dds_listener_t *src*)
Copy the listener callbacks from source to destination, unless already set.

Any listener callbacks already set in `dst` (including NULL) are skipped, only those set to DDS_LUNSET are copied from `src`.

Parameters

- `dst`: The pointer to the destination listener structure, where the content is merged
- `src`: The pointer to the source listener structure to be copied

void **dds_listener_merge**(*dds_listener_t *dst*, **const** *dds_listener_t *src*)

void **dds_lset_inconsistent_topic**(*dds_listener_t *listener*, *dds_on_inconsistent_topic_fn callback*)
Set the inconsistent_topic callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_liveliness_lost**(*dds_listener_t *listener*, *dds_on_liveliness_lost_fn callback*)
Set the liveliness_lost callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_offered_deadline_missed**(*dds_listener_t *listener*, *dds_on_offered_deadline_missed_fn callback*)
Set the offered_deadline_missed callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_offered_incompatible_qos**(*dds_listener_t *listener*, *dds_on_offered_incompatible_qos_fn callback*)

Set the offered_incompatible_qos callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_data_on_readers**(*dds_listener_t *listener*, *dds_on_data_on_readers_fn callback*)

Set the data_on_readers callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_sample_lost**(*dds_listener_t *listener*, *dds_on_sample_lost_fn callback*)

Set the sample_lost callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_data_available**(*dds_listener_t *listener*, *dds_on_data_available_fn callback*)

Set the data_available callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_sample_rejected**(*dds_listener_t *listener*, *dds_on_sample_rejected_fn callback*)

Set the sample_rejected callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_liveliness_changed**(*dds_listener_t *listener*, *dds_on_liveliness_changed_fn callback*)

Set the liveliness_changed callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_requested_deadline_missed**(*dds_listener_t *listener*, *dds_on_requested_deadline_missed_fn callback*)

Set the requested_deadline_missed callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_requested_incompatible_qos**(*dds_listener_t *listener*, *dds_on_requested_incompatible_qos_fn callback*)

Set the requested_incompatible_qos callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_publication_matched**(*dds_listener_t *listener*, *dds_on_publication_matched_fn callback*)

Set the publication_matched callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lset_subscription_matched**(*dds_listener_t *listener*, *dds_on_subscription_matched_fn callback*)

Set the subscription_matched callback in the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be set
- `callback`: The callback to set in the listener, can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_inconsistent_topic**(**const** *dds_listener_t *listener*, *dds_on_inconsistent_topic_fn *callback*)

Get the inconsistent_topic callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from

- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_liveliness_lost**(**const** *dds_listener_t* *listener, *dds_on_liveliness_lost_fn* *callback)

Get the liveliness_lost callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_offered_deadline_missed**(**const** *dds_listener_t* *listener, *dds_on_offered_deadline_missed_fn* *callback)

Get the offered_deadline_missed callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_offered_incompatible_qos**(**const** *dds_listener_t* *listener, *dds_on_offered_incompatible_qos_fn* *callback)

Get the offered_incompatible_qos callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_data_on_readers**(**const** *dds_listener_t* *listener, *dds_on_data_on_readers_fn* *callback)

Get the data_on_readers callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_sample_lost**(**const** *dds_listener_t* *listener, *dds_on_sample_lost_fn* *callback)

Get the sample_lost callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_data_available**(**const** *dds_listener_t* *listener, *dds_on_data_available_fn* *callback)

Get the data_available callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_sample_rejected**(**const** *dds_listener_t* *listener, *dds_on_sample_rejected_fn* *callback)

Get the sample_rejected callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_liveliness_changed**(**const** *dds_listener_t* *listener, *dds_on_liveliness_changed_fn* *callback)

Get the liveliness_changed callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_requested_deadline_missed**(**const** *dds_listener_t* *listener, *dds_on_requested_deadline_missed_fn* *callback)

Get the requested_deadline_missed callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_requested_incompatible_qos**(**const** *dds_listener_t* *listener, *dds_on_requested_incompatible_qos_fn* *callback)

Get the requested_incompatible_qos callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_publication_matched**(**const** *dds_listener_t* *listener, *dds_on_publication_matched_fn* *callback)

Get the publication_matched callback from the listener structure.

Parameters

- `listener`: The pointer to the listener structure, where the callback will be retrieved from
- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer

void **dds_lget_subscription_matched**(**const** *dds_listener_t* **listener*, *dds_on_subscription_matched_fn* **callback*)

Get the subscription_matched callback from the listener structure.

Parameters

- `callback`: Pointer where the retrieved callback can be stored; can be NULL, ::DDS_LUNSET or a valid callback pointer
- `listener`: The pointer to the listener structure, where the callback will be retrieved from

file **dds_public_qos.h**

#include "os/os_public.h"#include "ddsc/dds_export.h" DDS C QoS API.

This header file defines the public API of QoS and Policies in the Eclipse Cyclone DDS C language binding.

QoS identifiers

DDS_INVALID_QOS_POLICY_ID

DDS_USERDATA_QOS_POLICY_ID

DDS_DURABILITY_QOS_POLICY_ID

DDS_PRESENTATION_QOS_POLICY_ID

DDS_DEADLINE_QOS_POLICY_ID

DDS_LATENCYBUDGET_QOS_POLICY_ID

DDS_OWNERSHIP_QOS_POLICY_ID

DDS_OWNERSHIPSTRENGTH_QOS_POLICY_ID

DDS_LIVELINESS_QOS_POLICY_ID

DDS_TIMEBASEDFILTER_QOS_POLICY_ID

DDS_PARTITION_QOS_POLICY_ID

DDS_RELIABILITY_QOS_POLICY_ID

DDS_DESTINATIONORDER_QOS_POLICY_ID

DDS_HISTORY_QOS_POLICY_ID

DDS_RESOURCELIMITS_QOS_POLICY_ID

DDS_ENTITYFACTORY_QOS_POLICY_ID

DDS_WRITERDATALIFECYCLE_QOS_POLICY_ID

DDS_READERDATALIFECYCLE_QOS_POLICY_ID

DDS_TOPICDATA_QOS_POLICY_ID

DDS_GROUPDATA_QOS_POLICY_ID

DDS_TRANSPORTPRIORITY_QOS_POLICY_ID

DDS_LIFESPAN_QOS_POLICY_ID

DDS_DURABILITYSERVICE_QOS_POLICY_ID

Typedefs

typedef struct nn_xqos **dds_qos_t**
QoS structure

typedef enum *dds_durability_kind* **dds_durability_kind_t**
Durability QoS: Applies to Topic, DataReader, DataWriter

typedef enum *dds_history_kind* **dds_history_kind_t**
History QoS: Applies to Topic, DataReader, DataWriter

typedef enum *dds_ownership_kind* **dds_ownership_kind_t**
Ownership QoS: Applies to Topic, DataReader, DataWriter

typedef enum *dds_liveliness_kind* **dds_liveliness_kind_t**
Liveliness QoS: Applies to Topic, DataReader, DataWriter

typedef enum *dds_reliability_kind* **dds_reliability_kind_t**
Reliability QoS: Applies to Topic, DataReader, DataWriter

typedef enum *dds_destination_order_kind* **dds_destination_order_kind_t**
DestinationOrder QoS: Applies to Topic, DataReader, DataWriter

typedef struct *dds_history_qospolicy* **dds_history_qospolicy_t**
History QoS: Applies to Topic, DataReader, DataWriter

typedef struct *dds_resource_limits_qospolicy* **dds_resource_limits_qospolicy_t**
ResourceLimits QoS: Applies to Topic, DataReader, DataWriter

typedef enum *dds_presentation_access_scope_kind* **dds_presentation_access_scope_kind_t**
Presentation QoS: Applies to Publisher, Subscriber

Enums

enum dds_durability_kind
Durability QoS: Applies to Topic, DataReader, DataWriter

Values:

DDS_DURABILITY_VOLATILE

DDS_DURABILITY_TRANSIENT_LOCAL

DDS_DURABILITY_TRANSIENT

DDS_DURABILITY_PERSISTENT

enum dds_history_kind
History QoS: Applies to Topic, DataReader, DataWriter

Values:

DDS_HISTORY_KEEP_LAST

DDS_HISTORY_KEEP_ALL

enum dds_ownership_kind
Ownership QoS: Applies to Topic, DataReader, DataWriter

Values:

DDS_OWNERSHIP_SHARED

DDS_OWNERSHIP_EXCLUSIVE

enum dds_liveliness_kind
Liveliness QoS: Applies to Topic, DataReader, DataWriter

Values:

DDS_LIVELINESS_AUTOMATIC

DDS_LIVELINESS_MANUAL_BY_PARTICIPANT

DDS_LIVELINESS_MANUAL_BY_TOPIC

enum dds_reliability_kind
Reliability QoS: Applies to Topic, DataReader, DataWriter

Values:

DDS_RELIABILITY_BEST_EFFORT

DDS_RELIABILITY_RELIABLE

enum dds_destination_order_kind
DestinationOrder QoS: Applies to Topic, DataReader, DataWriter

Values:

DDS_DESTINATIONORDER_BY_RECEPTION_TIMESTAMP

DDS_DESTINATIONORDER_BY_SOURCE_TIMESTAMP

enum dds_presentation_access_scope_kind
Presentation QoS: Applies to Publisher, Subscriber

Values:

DDS_PRESENTATION_INSTANCE

DDS_PRESENTATION_TOPIC

DDS_PRESENTATION_GROUP

Functions

dds_qos_t ***dds_create_qos**(void)
Allocate memory and initialize default QoS-policies.

Return - Pointer to the initialized dds_qos_t structure, NULL if unsuccessful.

dds_qos_t ***dds_qos_create**(void)

void **dds_delete_qos**(*dds_qos_t* **qos*)
Delete memory allocated to QoS-policies structure.

Parameters

- `qos`: - Pointer to dds_qos_t structure

void **dds_qos_delete** (*dds_qos_t *qos*)

void **dds_reset_qos** (*dds_qos_t *qos*)
Reset a QoS-policies structure to default values.

Parameters

- `qos`: - Pointer to the dds_qos_t structure

void **dds_qos_reset** (*dds_qos_t *qos*)

dds_return_t **dds_copy_qos** (*dds_qos_t *dst*, **const** *dds_qos_t *src*)
Copy all QoS-policies from one structure to another.

Return - Return-code indicating success or failure

Parameters

- `dst`: - Pointer to the destination dds_qos_t structure
- `src`: - Pointer to the source dds_qos_t structure

dds_return_t **dds_qos_copy** (*dds_qos_t *dst*, **const** *dds_qos_t *src*)

void **dds_merge_qos** (*dds_qos_t *dst*, **const** *dds_qos_t *src*)
Copy all QoS-policies from one structure to another, unless already set.

Policies are copied from src to dst, unless src already has the policy set to a non-default value.

Parameters

- `dst`: - Pointer to the destination qos structure
- `src`: - Pointer to the source qos structure

void **dds_qos_merge** (*dds_qos_t *dst*, **const** *dds_qos_t *src*)

bool **dds_qos_equal** (**const** *dds_qos_t *a*, **const** *dds_qos_t *b*)
Copy all QoS-policies from one structure to another, unless already set.

Policies are copied from src to dst, unless src already has the policy set to a non-default value.

Parameters

- `dst`: - Pointer to the destination qos structure
- `src`: - Pointer to the source qos structure

void **dds_qset_userdata** (*dds_qos_t *qos*, **const** void *value*, size_t *sz*)
Set the userdata of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the userdata
- `value`: - Pointer to the userdata
- `sz`: - Size of userdata stored in value

void **dds_qset_topicdata** (*dds_qos_t *qos*, **const** void *value*, size_t *sz*)
Set the topicdata of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the topicdata
- `value`: - Pointer to the topicdata
- `sz`: - Size of the topicdata stored in value

void **dds_qset_groupdata**(*dds_qos_t* *qos, **const** void *value, size_t sz)
Set the groupdata of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the groupdata
- `value`: - Pointer to the group data
- `sz`: - Size of groupdata stored in value

void **dds_qset_durability**(*dds_qos_t* *qos, *dds_durability_kind_t* kind)
Set the durability policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `kind`: - Durability kind value DCPS_QoS_Durability

void **dds_qset_history**(*dds_qos_t* *qos, *dds_history_kind_t* kind, int32_t depth)
Set the history policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `kind`: - History kind value DCPS_QoS_History
- `depth`: - History depth value DCPS_QoS_History

void **dds_qset_resource_limits**(*dds_qos_t* *qos, int32_t max_samples, int32_t max_instances, int32_t max_samples_per_instance)
Set the resource limits policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `max_samples`: - Number of samples resource-limit value
- `max_instances`: - Number of instances resource-limit value
- `max_samples_per_instance`: - Number of samples per instance resource-limit value

void **dds_qset_presentation**(*dds_qos_t* *qos, *dds_presentation_access_scope_kind_t* access_scope, bool coherent_access, bool ordered_access)
Set the presentation policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `access_scope`: - Access-scope kind

- `coherent_access`: - Coherent access enable value
- `ordered_access`: - Ordered access enable value

void **dds_qset_lifespan**(*dds_qos_t *qos*, *dds_duration_t lifespan*)
Set the lifespan policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `lifespan`: - Lifespan duration (expiration time relative to source timestamp of a sample)

void **dds_qset_deadline**(*dds_qos_t *qos*, *dds_duration_t deadline*)
Set the deadline policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `deadline`: - Deadline duration

void **dds_qset_latency_budget**(*dds_qos_t *qos*, *dds_duration_t duration*)
Set the latency-budget policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `duration`: - Latency budget duration

void **dds_qset_ownership**(*dds_qos_t *qos*, *dds_ownership_kind_t kind*)
Set the ownership policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `kind`: - Ownership kind

void **dds_qset_ownership_strength**(*dds_qos_t *qos*, int32_t *value*)
Set the ownership strength policy of a qos structure.

param[in,out] qos - Pointer to a dds_qos_t structure that will store the policy param[in] value - Ownership strength value

void **dds_qset_liveliness**(*dds_qos_t *qos*, *dds_liveliness_kind_t kind*, *dds_duration_t lease_duration*)
Set the liveliness policy of a qos structure.

param[in,out] qos - Pointer to a dds_qos_t structure that will store the policy param[in] kind - Liveliness kind param[in[lease_duration - Lease duration

void **dds_qset_time_based_filter**(*dds_qos_t *qos*, *dds_duration_t minimum_separation*)
Set the time-based filter policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `minimum_separation`: - Minimum duration between sample delivery for an instance

void **dds_qset_partition**(*dds_qos_t* *qos, uint32_t *n*, **const** char **ps)
Set the partition policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `n`: - Number of partitions stored in ps
- `[in[`: ps - Pointer to string(s) storing partition name(s)

void **dds_qset_reliability**(*dds_qos_t* *qos, *dds_reliability_kind_t* *kind*, *dds_duration_t* *max_blocking_time*)
Set the reliability policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `kind`: - Reliability kind
- `max_blocking_time`: - Max blocking duration applied when kind is reliable.

void **dds_qset_transport_priority**(*dds_qos_t* *qos, int32_t *value*)
Set the transport-priority policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `value`: - Priority value

void **dds_qset_destination_order**(*dds_qos_t* *qos, *dds_destination_order_kind_t* *kind*)
Set the destination-order policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `kind`: - Destination-order kind

void **dds_qset_writer_data_lifecycle**(*dds_qos_t* *qos, bool *autodispose*)
Set the writer data-lifecycle policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `autodispose_unregistered_instances`: - Automatic disposal of unregistered instances

void **dds_qset_reader_data_lifecycle**(*dds_qos_t* *qos, *dds_duration_t* *autopurge_nowriter_samples_delay*, *dds_duration_t* *autopurge_disposed_samples_delay*)
Set the reader data-lifecycle policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy

- `autopurge_nowriter_samples_delay`: - Delay for purging of samples from instances in a no-writers state
- `autopurge_disposed_samples_delay`: - Delay for purging of samples from disposed instances

void **dds_qset_durability_service**(*dds_qos_t* *qos, *dds_duration_t* service_cleanup_delay, *dds_history_kind_t* history_kind, int32_t history_depth, int32_t max_samples, int32_t max_instances, int32_t max_samples_per_instance)

Set the durability-service policy of a qos structure.

Parameters

- `qos`: - Pointer to a dds_qos_t structure that will store the policy
- `service_cleanup_delay`: - Delay for purging of abandoned instances from the durability service
- `history_kind`: - History policy kind applied by the durability service
- `history_depth`: - History policy depth applied by the durability service
- `max_samples`: - Number of samples resource-limit policy applied by the durability service
- `max_instances`: - Number of instances resource-limit policy applied by the durability service
- `max_samples_per_instance`: - Number of samples per instance resource-limit policy applied by the durability service

bool **dds_qget_userdata**(**const** *dds_qos_t* *qos, void **value, size_t *sz)

Get the userdata from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `value`: - Pointer that will store the userdata
- `sz`: - Pointer that will store the size of userdata

bool **dds_qget_topicdata**(**const** *dds_qos_t* *qos, void **value, size_t *sz)

Get the topicdata from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `value`: - Pointer that will store the topicdata
- `sz`: - Pointer that will store the size of topicdata

bool **dds_qget_groupdata**(**const** *dds_qos_t* *qos, void **value, size_t *sz)

Get the groupdata from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `value`: - Pointer that will store the groupdata
- `sz`: - Pointer that will store the size of groupdata

bool **dds_qget_durability**(**const** *dds_qos_t* *qos, *dds_durability_kind_t* *kind)
Get the durability policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `kind`: - Pointer that will store the durability kind

bool **dds_qget_history**(**const** *dds_qos_t* *qos, *dds_history_kind_t* *kind, int32_t *depth)
Get the history policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `kind`: - Pointer that will store the history kind (optional)
- `depth`: - Pointer that will store the history depth (optional)

bool **dds_qget_resource_limits**(**const** *dds_qos_t* *qos, int32_t *max_samples, int32_t *max_instances, int32_t *max_samples_per_instance)
Get the resource-limits policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `max_samples`: - Pointer that will store the number of samples resource-limit (optional)
- `max_instances`: - Pointer that will store the number of instances resource-limit (optional)
- `max_samples_per_instance`: - Pointer that will store the number of samples per instance resource-limit (optional)

bool **dds_qget_presentation**(**const** *dds_qos_t* *qos, *dds_presentation_access_scope_kind_t* *access_scope, bool *coherent_access, bool *ordered_access)
Get the presentation policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `access_scope`: - Pointer that will store access scope kind (optional)
- `coherent_access`: - Pointer that will store coherent access enable value (optional)
- `ordered_access`: - Pointer that will store orderede access enable value (optional)

bool **dds_qget_lifespan**(**const** *dds_qos_t *qos*, *dds_duration_t *lifespan*)
Get the lifespan policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `lifespan`: - Pointer that will store lifespan duration

bool **dds_qget_deadline**(**const** *dds_qos_t *qos*, *dds_duration_t *deadline*)
Get the deadline policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `deadline`: - Pointer that will store deadline duration

bool **dds_qget_latency_budget**(**const** *dds_qos_t *qos*, *dds_duration_t *duration*)
Get the latency-budget policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `duration`: - Pointer that will store latency-budget duration

bool **dds_qget_ownership**(**const** *dds_qos_t *qos*, *dds_ownership_kind_t *kind*)
Get the ownership policy from a qos structure.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `kind`: - Pointer that will store the ownership kind

bool **dds_qget_ownership_strength**(**const** *dds_qos_t *qos*, int32_t *value*)
Get the ownership strength qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `value`: - Pointer that will store the ownership strength value

bool **dds_qget_liveliness**(**const** *dds_qos_t *qos*, *dds_liveliness_kind_t *kind*, *dds_duration_t *lease_duration*)
Get the liveliness qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `kind`: - Pointer that will store the liveliness kind (optional)
- `lease_duration`: - Pointer that will store the liveliness lease duration (optional)

bool **dds_qget_time_based_filter**(**const** *dds_qos_t* *qos, *dds_duration_t* *minimum_separation)

Get the time-based filter qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `minimum_separation`: - Pointer that will store the minimum separation duration (optional)

bool **dds_qget_partition**(**const** *dds_qos_t* *qos, uint32_t *n, char ***ps)

Get the partition qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `n`: - Pointer that will store the number of partitions (optional)
- `ps`: - Pointer that will store the string(s) containing partition name(s) (optional)

bool **dds_qget_reliability**(**const** *dds_qos_t* *qos, *dds_reliability_kind_t* *kind, *dds_duration_t* *max_blocking_time)

Get the reliability qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `kind`: - Pointer that will store the reliability kind (optional)
- `max_blocking_time`: - Pointer that will store the max blocking time for reliable reliability (optional)

bool **dds_qget_transport_priority**(**const** *dds_qos_t* *qos, int32_t *value)

Get the transport priority qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `value`: - Pointer that will store the transport priority value

bool **dds_qget_destination_order**(**const** *dds_qos_t* *qos, *dds_destination_order_kind_t* *kind)

Get the destination-order qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `kind`: - Pointer that will store the destination-order kind

bool **dds_qget_writer_data_lifecycle**(**const** *dds_qos_t* *qos, bool *autodispose)
Get the writer data-lifecycle qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `autodispose_unregistered_instances`: - Pointer that will store the autodispose unregistered instances enable value

bool **dds_qget_reader_data_lifecycle**(**const** *dds_qos_t* *qos, *dds_duration_t* *autopurge_nowriter_samples_delay, *dds_duration_t* *autopurge_disposed_samples_delay)
Get the reader data-lifecycle qos policy.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `autopurge_nowriter_samples_delay`: - Pointer that will store the delay for autopurging samples from instances in a no-writer state (optional)
- `autopurge_disposed_samples_delay`: - Pointer that will store the delay for autopurging of disposed instances (optional)

bool **dds_qget_durability_service**(**const** *dds_qos_t* *qos, *dds_duration_t* *service_cleanup_delay, *dds_history_kind_t* *history_kind, int32_t *history_depth, int32_t *max_samples, int32_t *max_instances, int32_t *max_samples_per_instance)
Get the durability-service qos policy values.

Return - false iff any of the arguments is invalid or the qos is not present in the qos object

Parameters

- `qos`: - Pointer to a dds_qos_t structure storing the policy
- `service_cleanup_delay`: - Pointer that will store the delay for purging of abandoned instances from the durability service (optional)
- `history_kind`: - Pointer that will store history policy kind applied by the durability service (optional)
- `history_depth`: - Pointer that will store history policy depth applied by the durability service (optional)
- `max_samples`: - Pointer that will store number of samples resource-limit policy applied by the durability service (optional)
- `max_instances`: - Pointer that will store number of instances resource-limit policy applied by the durability service (optional)

- `max_samples_per_instance`: - Pointer that will store number of samples per instance resource-limit policy applied by the durability service (optional)

file **dds_public_status.h**

#include "os/os_public.h"#include "ddsc/dds_export.h" DDS C Communication Status API.

This header file defines the public API of the Communication Status in the Eclipse Cyclone DDS C language binding.

Typedefs

typedef struct *dds_offered_deadline_missed_status* **dds_offered_deadline_missed_status_t**
DCPS_Status_OfferedDeadlineMissed

typedef struct *dds_offered_incompatible_qos_status* **dds_offered_incompatible_qos_status_t**
DCPS_Status_OfferedIncompatibleQoS

typedef struct *dds_publication_matched_status* **dds_publication_matched_status_t**
DCPS_Status_PublicationMatched

typedef struct *dds_liveliness_lost_status* **dds_liveliness_lost_status_t**
DCPS_Status_LivelinessLost

typedef struct *dds_subscription_matched_status* **dds_subscription_matched_status_t**
DCPS_Status_SubscriptionMatched

typedef struct *dds_sample_rejected_status* **dds_sample_rejected_status_t**
DCPS_Status_SampleRejected

typedef struct *dds_liveliness_changed_status* **dds_liveliness_changed_status_t**
DCPS_Status_LivelinessChanged

typedef struct *dds_requested_deadline_missed_status* **dds_requested_deadline_missed_status_t**
DCPS_Status_RequestedDeadlineMissed

typedef struct *dds_requested_incompatible_qos_status* **dds_requested_incompatible_qos_status_t**
DCPS_Status_RequestedIncompatibleQoS

typedef struct *dds_sample_lost_status* **dds_sample_lost_status_t**
DCPS_Status_SampleLost

typedef struct *dds_inconsistent_topic_status* **dds_inconsistent_topic_status_t**
DCPS_Status_InconsistentTopic

Enums

enum dds_sample_rejected_status_kind
dds_sample_rejected_status_kind

Values:

DDS_NOT_REJECTED

DDS_REJECTED_BY_INSTANCES_LIMIT

DDS_REJECTED_BY_SAMPLES_LIMIT

DDS_REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT

Functions

dds_return_t **dds_get_inconsistent_topic_status** (*dds_entity_t topic*, *dds_inconsistent_topic_status_t *status*)

Get INCONSISTENT_TOPIC status.

This operation gets the status value corresponding to INCONSISTENT_TOPIC and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `topic`: The entity to get the status
- `status`: The pointer to DCPS_Status_InconsistentTopic to get the status

Return Value

- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_publication_matched_status** (*dds_entity_t writer*, *dds_publication_matched_status_t *status*)

Get PUBLICATION_MATCHED status.

This operation gets the status value corresponding to PUBLICATION_MATCHED and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `writer`: The entity to get the status
- `status`: The pointer to DCPS_Status_PublicationMatched to get the status

Return Value

- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_liveliness_lost_status** (*dds_entity_t writer*, *dds_liveliness_lost_status_t *status*)

Get LIVELINESS_LOST status.

This operation gets the status value corresponding to LIVELINESS_LOST and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `writer`: The entity to get the status
- `status`: The pointer to DCPS_Status_LivelinessLost to get the status

Return Value

- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_offered_deadline_missed_status** (*dds_entity_t writer*, *dds_offered_deadline_missed_status_t *status*)

Get OFFERED_DEADLINE_MISSED status.

This operation gets the status value corresponding to OFFERED_DEADLINE_MISSED and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `writer`: The entity to get the status
- `status`: The pointer to DCPS_Status_OfferedDeadlineMissed to get the status

Return Value

- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_offered_incompatible_qos_status** (*dds_entity_t writer*, *dds_offered_incompatible_qos_status_t *status*)

Get OFFERED_INCOMPATIBLE_QOS status.

This operation gets the status value corresponding to OFFERED_INCOMPATIBLE_QOS and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `writer`: The writer entity to get the status
- `status`: The pointer to DCPS_Status_OfferedIncompatibleQoS to get the status

Return Value

- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_subscription_matched_status** (*dds_entity_t reader*, *dds_subscription_matched_status_t *status*)

Get SUBSCRIPTION_MATCHED status.

This operation gets the status value corresponding to SUBSCRIPTION_MATCHED and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `reader`: The reader entity to get the status
- `status`: The pointer to DCPS_Status_SubscriptionMatched to get the status

Return Value

- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_liveliness_changed_status** (*dds_entity_t reader*, *dds_liveliness_changed_status_t *status*)

Get LIVELINESS_CHANGED status.

This operation gets the status value corresponding to LIVELINESS_CHANGED and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- `reader`: The entity to get the status
- `status`: The pointer to DCPS_Status_LivelinessChanged to get the status

Return Value

- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: One of the given arguments is not valid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_return_t **dds_get_sample_rejected_status** (*dds_entity_t reader*, *dds_sample_rejected_status_t *status*)

Get SAMPLE_REJECTED status.

This operation gets the status value corresponding to SAMPLE_REJECTED and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return 0 - Success

Return <0 - Failure (use *dds_err_nr()* to get error value).

Parameters

- reader: The entity to get the status
- status: The pointer to DCPS_Status_SampleRejected to get the status

Return Value

- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: One of the given arguments is not valid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_return_t **dds_get_sample_lost_status** (*dds_entity_t reader*, *dds_sample_lost_status_t *status*)

Get SAMPLE_LOST status.

This operation gets the status value corresponding to SAMPLE_LOST and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return A dds_return_t indicating success or failure

Parameters

- reader: The entity to get the status
- status: The pointer to DCPS_Status_SampleLost to get the status

Return Value

- DDS_RETCODE_OK: Success
- DDS_RETCODE_ERROR: An internal error has occurred.
- DDS_RETCODE_BAD_PARAMETER: One of the given arguments is not valid.
- DDS_RETCODE_ILLEGAL_OPERATION: The operation is invoked on an inappropriate object.
- DDS_RETCODE_ALREADY_DELETED: The entity has already been deleted.

dds_return_t **dds_get_requested_deadline_missed_status**(*dds_entity_t reader*, *dds_requested_deadline_missed_status_t *status*)

Get REQUESTED_DEADLINE_MISSED status.

This operation gets the status value corresponding to REQUESTED_DEADLINE_MISSED and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return A dds_return_t indicating success or failure

Parameters

- `reader`: The entity to get the status
- `status`: The pointer to DCPS_Status_RequestedDeadlineMissed to get the status

Return Value

- `DDS_RETCODE_OK`: Success
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

dds_return_t **dds_get_requested_incompatible_qos_status**(*dds_entity_t reader*, *dds_requested_incompatible_qos_status_t *status*)

Get REQUESTED_INCOMPATIBLE_QOS status.

This operation gets the status value corresponding to REQUESTED_INCOMPATIBLE_QOS and reset the status. The value can be obtained, only if the status is enabled for an entity. NULL value for status is allowed and it will reset the trigger value when status is enabled.

Return A dds_return_t indicating success or failure

Parameters

- `reader`: The entity to get the status
- `status`: The pointer to DCPS_Status_RequestedIncompatibleQoS to get the status

Return Value

- `DDS_RETCODE_OK`: Success
- `DDS_RETCODE_ERROR`: An internal error has occurred.
- `DDS_RETCODE_BAD_PARAMETER`: One of the given arguments is not valid.
- `DDS_RETCODE_ILLEGAL_OPERATION`: The operation is invoked on an inappropriate object.
- `DDS_RETCODE_ALREADY_DELETED`: The entity has already been deleted.

file **dds_public_stream.h**

#include "os/os_public.h"#include <stdbool.h>#include "ddsc/dds_export.h" DDS C Stream API.

This header file defines the public API of the Streams in the Eclipse Cyclone DDS C language binding.

Defines

DDS_STREAM_BE

DDS_STREAM_LE

Typedefs

typedef struct *dds_stream* **dds_stream_t**

Functions

dds_stream_t ***dds_stream_create** (uint32_t *size*)

dds_stream_t ***dds_stream_from_buffer** (**const** void **buf*, size_t *sz*, int *bswap*)

void **dds_stream_delete** (*dds_stream_t* **st*)

void **dds_stream_fini** (*dds_stream_t* **st*)

void **dds_stream_reset** (*dds_stream_t* **st*)

void **dds_stream_init** (*dds_stream_t* **st*, uint32_t *size*)

void **dds_stream_grow** (*dds_stream_t* **st*, uint32_t *size*)

bool **dds_stream_endian** (void)

void **dds_stream_read_sample_w_desc** (*dds_stream_t* **is*, void **data*, **const struct** *dds_topic_descriptor* **desc*)

bool **dds_stream_read_bool** (*dds_stream_t* **is*)

uint8_t **dds_stream_read_uint8** (*dds_stream_t* **is*)

uint16_t **dds_stream_read_uint16** (*dds_stream_t* **is*)

uint32_t **dds_stream_read_uint32** (*dds_stream_t* **is*)

uint64_t **dds_stream_read_uint64** (*dds_stream_t* **is*)

float **dds_stream_read_float** (*dds_stream_t* **is*)

double **dds_stream_read_double** (*dds_stream_t* **is*)

char ***dds_stream_read_string** (*dds_stream_t* **is*)

void **dds_stream_read_buffer** (*dds_stream_t* **is*, uint8_t **buffer*, uint32_t *len*)

char **dds_stream_read_char** (*dds_stream_t* **is*)

int8_t **dds_stream_read_int8** (*dds_stream_t* **is*)

int16_t **dds_stream_read_int16** (*dds_stream_t* **is*)

int32_t **dds_stream_read_int32** (*dds_stream_t* **is*)

int64_t **dds_stream_read_int64** (*dds_stream_t* **is*)

void **dds_stream_write_bool** (*dds_stream_t* **os*, bool *val*)

void **dds_stream_write_uint8** (*dds_stream_t* **os*, uint8_t *val*)

void **dds_stream_write_uint16** (*dds_stream_t* **os*, uint16_t *val*)

void **dds_stream_write_uint32** (*dds_stream_t* **os*, uint32_t *val*)

void **dds_stream_write_uint64** (*dds_stream_t* *os, uint64_t *val*)

void **dds_stream_write_float** (*dds_stream_t* *os, float *val*)

void **dds_stream_write_double** (*dds_stream_t* *os, double *val*)

void **dds_stream_write_string** (*dds_stream_t* *os, **const** char *val*)

void **dds_stream_write_buffer** (*dds_stream_t* *os, uint32_t *len*, **const** uint8_t *buffer*)

void ***dds_stream_address** (*dds_stream_t* *s*)

void ***dds_stream_alignto** (*dds_stream_t* *s, uint32_t *a*)

void **dds_stream_write_char** (*dds_stream_t* *os, char *val*)

void **dds_stream_write_int8** (*dds_stream_t* *os, int8_t *val*)

void **dds_stream_write_int16** (*dds_stream_t* *os, int16_t *val*)

void **dds_stream_write_int32** (*dds_stream_t* *os, int32_t *val*)

void **dds_stream_write_int64** (*dds_stream_t* *os, int64_t *val*)

file **dds_public_time.h**

#include "os/os_public.h"#include "ddsc/dds_export.h" DDS C Time support API.

This header file defines the public API of the in the Eclipse Cyclone DDS C language binding.

Macro definition for time units in nanoseconds.

DDS_NSECS_IN_SEC

DDS_NSECS_IN_MSEC

DDS_NSECS_IN_USEC

Infinite timeout for indicate absolute time

DDS_NEVER

Infinite timeout for relative time

DDS_INFINITY

Macro definition for time conversion from nanoseconds

DDS_SECS (n)

DDS_MSECS (n)

DDS_USECS (n)

Typedefs

typedef int64_t **dds_time_t**
Absolute Time definition

typedef int64_t **dds_duration_t**
Relative Time definition

Functions

dds_time_t **dds_time** (void)
Description : This operation returns the current time (in nanoseconds)

Arguments :

1. Returns current time

void **dds_sleepfor** (*dds_duration_t n*)
Description : This operation blocks the calling thread until the relative time n has elapsed

Arguments :

1. n Relative Time to block a thread

void **dds_sleepuntil** (*dds_time_t n*)
Description : This operation blocks the calling thread until the absolute time n has elapsed

Arguments :

1. n absolute Time to block a thread

dir **/home/erik/cyclonedds/src/core**

dir **/home/erik/cyclonedds/src/core/ddsc/include/ddsc**

dir **/home/erik/cyclonedds/src/core/ddsc**

dir **/home/erik/cyclonedds/src/core/ddsc/include**

dir **/home/erik/cyclonedds/src**

CHAPTER 7

A guide to the configuration options of Eclipse Cyclone DDS

This document attempts to provide background information that will help in adjusting the configuration of Eclipse Cyclone DDS when the default settings do not give the desired behavior. A full listing of all settings is out of scope for this document, but can be extracted from the sources.

7.1 DDSI Concepts

The DDSI standard is intimately related to the DDS 1.2 and 1.4 standards, with a clear correspondence between the entities in DDSI and those in DCPS. However, this correspondence is not one-to-one.

In this section we give a high-level description of the concepts of the DDSI specification, with hardly any reference to the specifics of the Eclipse Cyclone DDS implementation, which are addressed in subsequent sections. This division was chosen to aid readers interested in interoperability to understand where the specification ends and the Eclipse Cyclone DDS implementation begins.

7.1.1 Mapping of DCPS domains to DDSI domains

In DCPS, a domain is uniquely identified by a non-negative integer, the domain id. In the UDP/IP mapping, this domain id is mapped to port numbers to be used for communicating with the peer nodes — these port numbers are particularly important for the discovery protocol — and this mapping of domain ids to UDP/IP port numbers ensures that accidental cross-domain communication is impossible with the default mapping.

DDSI does not communicate the DCPS port number in the discovery protocol; it assumes that each domain id maps to a unique port number. While it is unusual to change the mapping, the specification requires this to be possible, and this means that two different DCPS domain ids can be mapped to a single DDSI domain.

7.1.2 Mapping of DCPS entities to DDSI entities

Each DCPS domain participant in a domain is mirrored in DDSI as a DDSI participant. These DDSI participants drive the discovery of participants, readers and writers in DDSI via the discovery protocols. By default, each DDSI participant has a unique address on the network in the form of its own UDP/IP socket with a unique port number.

Any data reader or data writer created by a DCPS domain participant is mirrored in DDSI as a DDSI reader or writer. In this translation, some of the structure of the DCPS domain is obscured because the standardized parts of DDSI have no knowledge of DCPS Subscribers and Publishers. Instead, each DDSI reader is the combination of the corresponding DCPS data reader and the DCPS subscriber it belongs to; similarly, each DDSI writer is a combination of the corresponding DCPS data writer and DCPS publisher. This corresponds to the way the standardized DCPS built-in topics describe the DCPS data readers and data writers, as there are no standardized built-in topics for describing the DCPS subscribers and publishers either. Implementations can (and do) offer additional built-in topics for describing these entities and include them in the discovery, but these are non-standard extensions.

In addition to the application-created readers and writers (referred to as *endpoints*), DDSI participants have a number of DDSI built-in endpoints used for discovery and liveliness checking/asserting. The most important ones are those absolutely required for discovery: readers and writers for the discovery data concerning DDSI participants, DDSI readers and DDSI writers. Some other ones exist as well, and a DDSI implementation can leave out some of these if it has no use for them. For example, if a participant has no writers, it doesn't strictly need the DDSI built-in endpoints for describing writers, nor the DDSI built-in endpoint for learning of readers of other participants.

7.1.3 Reliable communication

Best-effort communication is simply a wrapper around UDP/IP: the packet(s) containing a sample are sent to the addresses at which the readers reside. No state is maintained on the writer. If a packet is lost, the reader will simply ignore the whatever samples were contained in the lost packet and continue with the next one.

When *reliable* communication is used, the writer does maintain a copy of the sample, in case a reader detects it has lost packets and requests a retransmission. These copies are stored in the writer history cache (or *WHC*) of the DDSI writer. The DDSI writer is required to periodically send *Heartbeats* to its readers to ensure that all readers will learn of the presence of new samples in the WHC even when packets get lost. It is allowed to suppress these periodic Heartbeats if there is all samples in the WHC have been acknowledged by all matched readers and the Eclipse Cyclone DDS exploits this freedom.

If a reader receives a Heartbeat and detects it did not receive all samples, it requests a retransmission by sending an *AckNack* message to the writer. The timing of this is somewhat adjustable and it is worth remarking that a roundtrip latency longer than the Heartbeat interval easily results in multiple retransmit requests for a single sample. In addition to requesting retransmission of some samples, a reader also uses the AckNack messages to inform the writer up to what sample it has received everything, and which ones it has not yet received. Whenever the writer indicates it requires a response to a Heartbeat the readers will send an AckNack message even when no samples are missing. In this case, it becomes a pure acknowledgement.

The combination of these behaviours in principle allows the writer to remove old samples from its WHC when it fills up too far, and allows readers to always receive all data. A complication exists in the case of unresponsive readers, readers that do not respond to a Heartbeat at all, or that for some reason fail to receive some samples despite resending it. The specification leaves the way these get treated unspecified. The default beahviour of Eclipse Cyclone DDS is to never consider readers unresponsive, but it can be configured to consider them so after a certain length of time has passed at which point the participant containing the reader is undiscovered.

Note that while this Heartbeat/AckNack mechanism is very straightforward, the specification actually allows suppressing heartbeats, merging of AckNacks and retransmissions, etc. The use of these techniques is required to allow for a performant DDSI implementation, whilst avoiding the need for sending redundant messages.

7.1.4 DDSI-specific transient-local behaviour

The above describes the essentials of the mechanism used for samples of the *volatile* durability kind, but the DCPS specification also provides *transient-local*, *transient* and *persistent* data. Of these, the DDSI specification at present only covers *transient-local*, and this is the only form of durable data available when interoperating across vendors.

In DDSI, transient-local data is implemented using the WHC that is normally used for reliable communication. For transient-local data, samples are retained even when all readers have acknowledged them. With the default history setting of `KEEP_LAST` with `history_depth = 1`, this means that late-joining readers can still obtain the latest sample for each existing instance.

Naturally, once the DCPS writer is deleted (or disappears for whatever reason), the DDSI writer disappears as well, and with it, its history. For this reason, transient data is generally much to be preferred over transient-local data. Eclipse Cyclone DDS has a facility for retrieving transient data from an suitably configured OpenSplice node, but does not yet include a native service for managing transient data.

7.1.5 Discovery of participants & endpoints

DDSI participants discover each other by means of the *Simple Participant Discovery Protocol* or *SPDP* for short. This protocol is based on periodically sending a message containing the specifics of the participant to a set of known addresses. By default, this is a standardised multicast address (`239.255.0.1`; the port number is derived from the domain id) that all DDSI implementations listen to.

Particularly important in the SPDP message are the unicast and multicast addresses at which the participant can be reached. Typically, each participant has a unique unicast address, which in practice means all participants on a node all have a different UDP/IP port number in their unicast address. In a multicast-capable network, it doesn't matter what the actual address (including port number) is, because all participants will learn them through these SPDP messages.

The protocol does allow for unicast-based discovery, which requires listing the addresses of machines where participants may be located and ensuring each participant uses one of a small set of port numbers. Because of this, some of the port numbers are derived not only from the domain id, but also from a *participant index*, which is a small non-negative integer, unique to a participant within a node. (Eclipse Cyclone DDS adds an indirection and uses at most one participant index for a domain for each process, regardless of how many DCPS participants are created by the process.)

Once two participants have discovered each other and both have matched the DDSI built-in endpoints their peer is advertising in the SPDP message, the *Simple Endpoint Discovery Protocol* or *SEDP* takes over, exchanging information on the DCPS data readers and data writers (and for Eclipse Cyclone DDS, also publishers, subscribers and topics in a manner compatible with OpenSplice) in the two participants.

The SEDP data is handled as reliable, transient-local data. Therefore, the SEDP writers send Heartbeats, the SEDP readers detect they have not yet received all samples and send AckNacks requesting retransmissions, the writer responds to these and eventually receives a pure acknowledgement informing it that the reader has now received the complete set.

Note that the discovery process necessarily creates a burst of traffic each time a participant is added to the system: *all* existing participants respond to the SPDP message, following which all start exchanging SEDP data.

7.2 Eclipse Cyclone DDS specifics

7.2.1 Discovery behaviour

Proxy participants and endpoints

Eclipse Cyclone DDS is what the DDSI specification calls a *stateful* implementation. Writers only send data to discovered readers and readers only accept data from discovered writers. (There is one exception: the writer may choose to multicast the data, and anyone listening will be able to receive it, if a reader has already discovered the writer but not vice-versa; it may accept the data even though the connection is not fully established yet. At present, not only can such asymmetrical discovery cause data to be delivered when it was perhaps not expected, it can also cause indefinite blocking if the situation persists for a long time.) Consequently, for each remote participant and reader

or writer, Eclipse Cyclone DDS internally creates a proxy participant, proxy reader or proxy writer. In the discovery process, writers are matched with proxy readers, and readers are matched with proxy writers, based on the topic and type names and the QoS settings.

Proxies have the same natural hierarchy that 'normal' DDSI entities have: each proxy endpoint is owned by some proxy participant, and once the proxy participant is deleted, all of its proxy endpoints are deleted as well. Participants assert their liveliness periodically (called *automic* liveliness in the DCPS specification and the only mode currently supported by Eclipse Cyclone DDS), and when nothing has been heard from a participant for the lease duration published by that participant in its SPDP message, the lease becomes expired triggering a clean-up.

Under normal circumstances, deleting endpoints simply triggers disposes and unregisters in SEDP protocol, and, similarly, deleting a participant also creates special messages that allow the peers to immediately reclaim resources instead of waiting for the lease to expire.

Sharing of discovery information

As Eclipse Cyclone DDS handles any number of participants in an integrated manner, the discovery protocol as sketched earlier is rather wasteful: there is no need for each individual participant in a Eclipse Cyclone DDS process to run the full discovery protocol for itself.

Instead of implementing the protocol as suggested by the standard, Eclipse Cyclone DDS shares all discovery activities amongst the participants, allowing one to add participants on a process with only a minimal impact on the system. It is even possible to have only a single DDSI participant in a process regardless of the number of DCPS participants created by the application code in that process, which then becomes the virtual owner of all the endpoints created in that one process. (See *Combining multiple participants*.) In this latter mode, there is no discovery penalty at all for having many participants, but evidently, any participant-based liveliness monitoring will be affected.

Because other implementations of the DDSI specification may be written on the assumption that all participants perform their own discovery, it is possible to simulate that with Eclipse Cyclone DDS. It will not actually perform the discovery for each participant independently, but it will generate the network traffic *as if* it does. These are controlled by the `Internal/BuiltinEndpointSet` and `Internal/ConservativeBuiltinReaderStartup` options. However, please note that at the time of writing, we are not aware of any DDSI implementation requiring the use of these settings.)

By sharing the discovery information across all participants in a single node, each new participant or endpoint is immediately aware of the existing peers and will immediately try to communicate with these peers. This may generate some redundant network traffic if these peers take a significant amount of time for discovering this new participant or endpoint.

Lingering writers

When an application deletes a reliable DCPS data writer, there is no guarantee that all its readers have already acknowledged the correct receipt of all samples. In such a case, Eclipse Cyclone DDS lets the writer (and the owning participant if necessary) linger in the system for some time, controlled by the `Internal/WriterLingerDuration` option. The writer is deleted when all samples have been acknowledged by all readers or the linger duration has elapsed, whichever comes first.

Note that the writer linger duration setting is currently not applied when Eclipse Cyclone DDS is requested to terminate.

Start-up mode

A similar issue exists when starting Eclipse Cyclone DDS: DDSI discovery takes time, and when data is written immediately after the first participant was created, it is likely that the discovery process hasn't completed yet and some remote readers have not yet been discovered. This would cause the writers to throw away samples for lack of interest,

even though matching readers already existed at the time of starting. For best-effort writers, this is perhaps surprising but still acceptable; for reliable writers, however, it would be very counter-intuitive.

Hence the existence of the so-called *start-up mode*, during which all volatile reliable writers are treated as-if they are transient-local writers. Transient-local data is meant to ensure samples are available to late-joining readers, the start-up mode uses this same mechanism to ensure late-discovered readers will also receive the data. This treatment of volatile data as-if it were transient-local happens internally and is invisible to the outside world, other than the availability of some samples that would not otherwise be available.

Once initial discovery has been completed, any new local writers can be matched locally against already existing readers, and consequently keeps any new samples published in a writer history cache because these existing readers have not acknowledged them yet. Hence why this mode is tied to the start-up of the DDSI stack, rather than to that of an individual writer.

Unfortunately it is impossible to detect with certainty when the initial discovery process has been completed and therefore the duration of this start-up mode is controlled by an option: `General/StartupModeDuration`.

While in general this start-up mode is beneficial, it is not always so. There are two downsides: the first is that during the start-up period, the writer history caches can grow significantly larger than one would normally expect; the second is that it does mean large amounts of historical data may be transferred to readers discovered relatively late in the process.

7.2.2 Writer history QoS and throttling

The DDSI specification heavily relies on the notion of a writer history cache (WHC) within which a sequence number uniquely identifies each sample. This WHC integrates two different indices on the samples published by a writer: one is on sequence number, used for retransmitting lost samples, and one is on key value and is used for retaining the current state of each instance in the WHC.

The index on key value allows dropping samples from the index on sequence number when the state of an instance is overwritten by a new sample. For transient-local, it conversely (also) allows retaining the current state of each instance even when all readers have acknowledged a sample.

The index on sequence number is required for retransmitting old data, and is therefore needed for all reliable writers. The index on key values is always needed for transient-local data, and will be default also be used for other writers using a history setting of `KEEP_LAST`. (The `Internal/AggressiveKeepLastWhc` setting controls this behaviour.) The advantage of an index on key value in such a case is that superseded samples can be dropped aggressively, instead of having to deliver them to all readers; the disadvantage is that it is somewhat more resource-intensive.

The WHC distinguishes between history to be retained for existing readers (controlled by the writer's history QoS setting) and the history to be retained for late-joining readers for transient-local writers (controlled by the topic's durability-service history QoS setting). This makes it possible to create a writer that never overwrites samples for live readers while maintaining only the most recent samples for late-joining readers. Moreover, it ensures that the data that is available for late-joining readers is the same for transient-local and for transient data.

Writer throttling is based on the WHC size using a simple controller. Once the WHC contains at least *high* bytes in unacknowledged samples, it stalls the writer until the number of bytes in unacknowledged samples drops below `Internal/Watermarks/WhcLow`. The value of *high* is dynamically adjusted between `Internal/Watermarks/WhcLow` and `Internal/Watermarks/WhcHigh` based on transmit pressure and receive retransmit requests. The initial value of *high* is `Internal/Watermarks/WhcHighInit` and the adaptive behavior can be disabled by setting `Internal/Watermarks/WhcAdaptive` to false.

While the adaptive behaviour generally handles a variety of fast and slow writers and readers quite well, the introduction of a very slow reader with small buffers in an existing network that is transmitting data at high rates can cause a sudden stop while the new reader tries to recover the large amount of data stored in the writer, before things can continue at a much lower rate.

7.3 Network and discovery configuration

7.3.1 Networking interfaces

Eclipse Cyclone DDS uses a single network interface, the *preferred* interface, for transmitting its multicast packets and advertises only the address corresponding to this interface in the DDSI discovery protocol.

To determine the default network interface, the eligible interfaces are ranked by quality and then selects the interface with the highest quality. If multiple interfaces are of the highest quality, it will select the first enumerated one. Eligible interfaces are those that are up and have the right kind of address family (IPv4 or IPv6). Priority is then determined as follows:

- interfaces with a non-link-local address are preferred over those with a link-local one;
- multicast-capable is preferred (see also `Internal/AssumeMulticastCapable`), or if none is available
- non-multicast capable but neither point-to-point, or if none is available
- point-to-point, or if none is available
- loopback

If this procedure doesn't select the desired interface automatically, it can be overridden by setting `General/NetworkInterfaceAddress` to either the name of the interface, the IP address of the host on the desired interface, or the network portion of the IP address of the host on the desired interface. An exact match on the address is always preferred and is the only option that allows selecting the desired one when multiple addresses are tied to a single interface.

The default address family is IPv4, setting General/UseIPv6 will change this to IPv6. Currently, Eclipse Cyclone DDS does not mix IPv4 and IPv6 addressing. Consequently, all DDSI participants in the network must use the same addressing mode. When interoperating, this behaviour is the same, i.e., it will look at either IPv4 or IPv6 addresses in the advertised address information in the SPDP and SEDP discovery protocols.

IPv6 link-local addresses are considered undesirable because they need to be published and received via the discovery mechanism, but there is in general no way to determine to which interface a received link-local address is related.

If IPv6 is requested and the preferred interface has a non-link-local address, Cyclone DDS will operate in a *global addressing* mode and will only consider discovered non-link-local addresses. In this mode, one can select any set of interface for listening to multicasts. Note that this behaviour is essentially identical to that when using IPv4, as IPv4 does not have the formal notion of address scopes that IPv6 has. If instead only a link-local address is available, Eclipse Cyclone DDS will run in a *link-local addressing* mode. In this mode it will accept any address in a discovery packet, assuming that a link-local address is valid on the preferred interface. To minimise the risk involved in this assumption, it only allows the preferred interface for listening to multicasts.

When a remote participant publishes multiple addresses in its SPDP message (or in SEDP messages, for that matter), it will select a single address to use for communicating with that participant. The address chosen is the first eligible one on the same network as the locally chosen interface, else one that is on a network corresponding to any of the other local interfaces, and finally simply the first one. Eligibility is determined in the same way as for network interfaces.

Multicasting

Eclipse Cyclone DDS allows configuring to what extent multicast (the regular, any-source multicast as well as source-specific multicast) is to be used:

- whether to use multicast for data communications,
- whether to use multicast for participant discovery,
- on which interfaces to listen for multicasts.

It is advised to allow multicasting to be used. However, if there are restrictions on the use of multicasting, or if the network reliability is dramatically different for multicast than for unicast, it may be attractive to disable multicast for normal communications. In this case, setting `General/AllowMulticast` to `false` will force the use of unicast communications for everything.

If at all possible, it is strongly advised to leave multicast-based participant discovery enabled, because that avoids having to specify a list of nodes to contact, and it furthermore reduces the network load considerably. Setting `General/AllowMulticast` to `spdp` will allow participant discovery via multicast while disabling multicast for everything else.

To disable incoming multicasts, or to control from which interfaces multicasts are to be accepted, one can use the `General/MulticastRecvInterfaceAddresses` setting. This allows listening on no interface, the preferred, all or a specific set of interfaces.

TCP support

The DDSI protocol is really a protocol designed for a transport providing connectionless, unreliable datagrams. However, there are times where TCP is the only practical network transport available (for example, across a WAN). Because of this, Eclipse Cyclone DDS can use TCP instead of UDP.

The differences in the model of operation between DDSI and TCP are quite large: DDSI is based on the notion of peers, whereas TCP communication is based on the notion of a session that is initiated by a 'client' and accepted by a 'server', and so TCP requires knowledge of the servers to connect to before the DDSI discovery protocol can exchange that information. The configuration of this is done in the same manner as for unicast-based UDP discovery.

TCP reliability is defined in terms of these sessions, but DDSI reliability is defined in terms of DDSI discovery and liveliness management. It is therefore possible that a TCP connection is (forcibly) closed while the remote endpoint is still considered alive. Following a reconnect the samples lost when the TCP connection was closed can be recovered via the normal DDSI reliability. This also means that the Heartbeats and AckNacks still need to be sent over a TCP connection, and consequently that DDSI flow-control occurs on top of TCP flow-control.

Another point worth noting is that connection establishment takes a potentially long time, and that giving up on a transmission to a failed or no-longer reachable host can also take a long time. These long delays can be visible at the application level at present.

TLS support

The TCP mode can be used in conjunction with TLS to provide mutual authentication and encryption. When TLS is enabled, plain TCP connections are no longer accepted or initiated.

Raw Ethernet support

As an additional option, on Linux, Eclipse Cyclone DDS can use a raw Ethernet network interface to communicate without a configured IP stack.

Discovery configuration

Discovery addresses

The DDSI discovery protocols, SPDP for the domain participants and SEDP for their endpoints, usually operate well without any explicit configuration. Indeed, the SEDP protocol never requires any configuration.

The SPDP protocol periodically sends, for each domain participant, an SPDP sample to a set of addresses, which by default contains just the multicast address, which is standardised for IPv4 (`239.255.0.1`) but not for

IPv6 (it uses `ff02::ffff:239.255.0.1`). The actual address can be overridden using the `Discovery/SPDPMulticastAddress` setting, which requires a valid multicast address.

In addition (or as an alternative) to the multicast-based discovery, any number of unicast addresses can be configured as addresses to be contacted by specifying peers in the `Discovery/Peers` section. Each time an SPDP message is sent, it is sent to all of these addresses.

Default behaviour is to include each IP address several times in the set (for participant indices 0 through `MaxAutoParticipantIndex`, each time with a different UDP port number (corresponding to another participant index), allowing at least several applications to be present on these hosts.

Obviously, configuring a number of peers in this way causes a large burst of packets to be sent each time an SPDP message is sent out, and each local DDSI participant causes a burst of its own. Most of the participant indices will not actually be use, making this rather wasteful behaviour.

To avoid sending large numbers of packets to each host, differing only in port number, it is also possible to add a port number to the IP address, formatted as IP:PORT, but this requires manually calculating the port number. In practice it also requires fixing the participant index using `Discovery/ParticipantIndex` (see the description of 'PI' in *Controlling port numbers*) to ensure that the configured port number indeed corresponds to the port number the remote DDSI implementation is listening on, and therefore is really attractive only when it is known that there is but a single DDSI process on that node.

Asymmetrical discovery

On reception of an SPDP packet, the addresses advertised in the packet are added to the set of addresses to which SPDP packets are sent periodically, allowing asymmetrical discovery. In an extreme example, if SPDP multicasting is disabled entirely, host A has the address of host B in its peer list and host B has an empty peer list, then B will eventually discover A because of an SPDP message sent by A, at which point it adds A's address to its own set and starts sending its own SPDP message to A, allowing A to discover B. This takes a bit longer than normal multicast based discovery, though, and risks writers being blocked by unresponsive readers.

Timing of SPDP packets

The interval with which the SPDP packets are transmitted is configurable as well, using the Discovery/SPDPInterval setting. A longer interval reduces the network load, but also increases the time discovery takes, especially in the face of temporary network disconnections.

Endpoint discovery

Although the SEDP protocol never requires any configuration, network partitioning does interact with it: so-called 'ignored partitions' can be used to instruct Eclipse Cyclone DDS to completely ignore certain DCPS topic and partition combinations, which will prevent data for these topic/partition combinations from being forwarded to and from the network.

7.3.2 Combining multiple participants

If a single process creates multiple participants, these are faithfully mirrored in DDSI participants and so a single process can appear as if it is a large system with many participants. The `Internal/SquashParticipants` option can be used to simulate the existence of only one participant, which owns all endpoints on that node. This reduces the background messages because far fewer liveliness assertions need to be sent, but there are some downsides.

Firstly, the liveliness monitoring features that are related to domain participants will be affected if multiple DCPS domain participants are combined into a single DDSI domain participant. For the 'automatic' liveliness setting, this is not an issue.

Secondly, this option makes it impossible for tooling to show the actual system topology.

Thirdly, the QoS of this sole participant is simply that of the first participant created in the process. In particular, no matter what other participants specify as their 'user data', it will not be visible on remote nodes.

There is an alternative that sits between squashing participants and normal operation, and that is setting `Internal/BuiltinEndpointSet` to `minimal`. In the default setting, each DDSI participant handled has its own writers for built-in topics and publishes discovery data on its own entities, but when set to 'minimal', only the first participant has these writers and publishes data on all entities. This is not fully compatible with other implementations as it means endpoint discovery data can be received for a participant that has not yet been discovered.

7.3.3 Controlling port numbers

The port numbers used by by Eclipse Cyclone DDS are determined as follows, where the first two items are given by the DDSI specification and the third is unique to Eclipse Cyclone DDS as a way of serving multiple participants by a single DDSI instance:

- 2 'well-known' multicast ports: `B` and `B+1`
- 2 unicast ports at which only this instance is listening: `B+PG*PI+10` and `B+PG*PI+11`
- 1 unicast port per domain participant it serves, chosen by the kernel from the anonymous ports, *i.e.* >= 32768

where:

- *B* is `Discovery/Ports/Base` (`7400`) + `Discovery/Ports/DomainGain` (`250`) * `Domain/Id`
- *PG* is `Discovery/Ports/ParticipantGain` (`2`)
- *PI* is `Discovery/ParticipantIndex`

The default values, taken from the DDSI specification, are in parentheses. There are actually even more parameters, here simply turned into constants as there is absolutely no point in ever changing these values; however, they *are* configurable and the interested reader is referred to the DDSI 2.1 or 2.2 specification, section 9.6.1.

PI is the most interesting, as it relates to having multiple processes in the same domain on a single node. Its configured value is either *auto*, *none* or a non-negative integer. This setting matters:

- When it is *auto* (which is the default), Eclipse Cyclone DDS probes UDP port numbers on start-up, starting with PI = 0, incrementing it by one each time until it finds a pair of available port numbers, or it hits the limit. The maximum PI it will ever choose is `Discovery/MaxAutoParticipantIndex` as a way of limiting the cost of unicast discovery.
- When it is *none* it simply ignores the 'participant index' altogether and asks the kernel to pick random ports (>= 32768). This eliminates the limit on the number of standalone deployments on a single machine and works just fine with multicast discovery while complying with all other parts of the specification for interoperability. However, it is incompatible with unicast discovery.
- When it is a non-negative integer, it is simply the value of PI in the above calculations. If multiple processes on a single machine are needed, they will need unique values for PI, and so for standalone deployments this particular alternative is hardly useful.

Clearly, to fully control port numbers, setting `Discovery/ParticipantIndex` (= PI) to a hard-coded value is the only possibility. By fixing PI, the port numbers needed for unicast discovery are fixed as well. This allows listing peers as IP:PORT pairs, significantly reducing traffic, as explained in the preceding subsection.

The other non-fixed ports that are used are the per-domain participant ports, the third item in the list. These are used only because there exist some DDSI implementations that assume each domain participant advertises a unique port

number as part of the discovery protocol, and hence that there is never any need for including an explicit destination participant id when intending to address a single domain participant by using its unicast locator. Eclipse Cyclone DDS never makes this assumption, instead opting to send a few bytes extra to ensure the contents of a message are all that is needed. With other implementations, you will need to check.

If all DDSI implementations in the network include full addressing information in the messages like Eclipse Cyclone DDS does, then the per-domain participant ports serve no purpose at all. The default `false` setting of `Compatibility/ManySocketsMode` disables the creation of these ports.

This setting can have a few other side benefits as well, as there will may be multiple DCPS participants using the same unicast locator. This improves the chances of a single unicast sufficing even when addressing a multiple participants.

7.4 Data path configuration

7.4.1 Retransmit merging

A remote reader can request retransmissions whenever it receives a Heartbeat and detects samples are missing. If a sample was lost on the network for many or all readers, the next heartbeat is likely to trigger a 'storm' of retransmission requests. Thus, the writer should attempt merging these requests into a multicast retransmission, to avoid retransmitting the same sample over & over again to many different readers. Similarly, while readers should try to avoid requesting retransmissions too often, in an interoperable system the writers should be robust against it.

In Eclipse Cyclone DDS, upon receiving a Heartbeat that indicates samples are missing, a reader will schedule the second and following retransmission requests to be sent after `Internal/NackDelay` or combine it with an already scheduled request if possible. Any samples received in between receipt of the Heartbeat and the sending of the AckNack will not need to be retransmitted.

Secondly, a writer attempts to combine retransmit requests in two different ways. The first is to change messages from unicast to multicast when another retransmit request arrives while the retransmit has not yet taken place. This is particularly effective when bandwidth limiting causes a backlog of samples to be retransmitted. The behaviour of the second can be configured using the `Internal/RetransmitMerging` setting. Based on this setting, a retransmit request for a sample is either honoured unconditionally, or it may be suppressed (or 'merged') if it comes in shortly after a multicasted retransmission of that very sample, on the assumption that the second reader will likely receive the retransmit, too. The `Internal/RetransmitMergingPeriod` controls the length of this time window.

7.4.2 Retransmit backlogs

Another issue is that a reader can request retransmission of many samples at once. When the writer simply queues all these samples for retransmission, it may well result in a huge backlog of samples to be retransmitted. As a result, the ones near the end of the queue may be delayed by so much that the reader issues another retransmit request.

Therefore, Eclipse Cyclone DDS limits the number of samples queued for retransmission and ignores (those parts of) retransmission requests that would cause the retransmit queue to contain too many samples or take too much time to process. There are two settings governing the size of these queues, and the limits are applied per timed-event thread. The first is `Internal/MaxQueuedRexmitMessages`, which limits the number of retransmit messages, the second `Internal/MaxQueuedRexmitBytes` which limits the number of bytes. The latter defaults to a setting based on the combination of the allowed transmit bandwidth and the `Internal/NackDelay` setting, as an approximation of the likely time until the next potential retransmit request from the reader.

7.4.3 Controlling fragmentation

Samples in DDS can be arbitrarily large, and will not always fit within a single datagram. DDSI has facilities to fragment samples so they can fit in UDP datagrams, and similarly IP has facilities to fragment UDP datagrams to

into network packets. The DDSI specification states that one must not unnecessarily fragment at the DDSI level, but Eclipse Cyclone DDS simply provides a fully configurable behaviour.

If the serialised form of a sample is at least `Internal/FragmentSize`, it will be fragmented using the DDSI fragmentation. All but the last fragment will be exactly this size; the last one may be smaller.

Control messages, non-fragmented samples, and sample fragments are all subject to packing into datagrams before sending it out on the network, based on various attributes such as the destination address, to reduce the number of network packets. This packing allows datagram payloads of up to `Internal/MaxMessageSize`, overshooting this size if the set maximum is too small to contain what must be sent as a single unit. Note that in this case, there is a real problem anyway, and it no longer matters where the data is rejected, if it is rejected at all. UDP/IP header sizes are not taken into account in this maximum message size.

The IP layer then takes this UDP datagram, possibly fragmenting it into multiple packets to stay within the maximum size the underlying network supports. A trade-off to be made is that while DDSI fragments can be retransmitted individually, the processing overhead of DDSI fragmentation is larger than that of UDP fragmentation.

7.4.4 Receive processing

Receiving of data is split into multiple threads:

- A single receive thread responsible for retrieving network packets and running the protocol state machine;
- A delivery thread dedicated to processing DDSI built-in data: participant discovery, endpoint discovery and liveliness assertions;
- One or more delivery threads dedicated to the handling of application data: deserialisation and delivery to the DCPS data reader caches.

The receive thread is responsible for retrieving all incoming network packets, running the protocol state machine, which involves scheduling of AckNack and Heartbeat messages and queueing of samples that must be retransmitted, and for defragmenting and ordering incoming samples.

Fragmented data first enters the defragmentation stage, which is per proxy writer. The number of samples that can be defragmented simultaneously is limited, for reliable data to `Internal/DefragReliableMaxSamples` and for unreliable data to `Internal/DefragUnreliableMaxSamples`.

Samples (defragmented if necessary) received out of sequence are buffered, primarily per proxy writer, but, secondarily, per reader catching up on historical (transient-local) data. The size of the first is limited to `Internal/PrimaryReorderMaxSamples`, the size of the second to `Internal/SecondaryReorderMaxSamples`.

In between the receive thread and the delivery threads sit queues, of which the maximum size is controlled by the `Internal/DeliveryQueueMaxSamples` setting. Generally there is no need for these queues to be very large (unless one has very small samples in very large messaegs), their primary function is to smooth out the processing when batches of samples become available at once, for example following a retransmission.

When any of these receive buffers hit their size limit and it concerns application data, the receive thread of will wait for the queue to shrink (a compromise that is the lesser evil within the constraints of various other choices). However, discovery data will never block the receive thread.

7.4.5 Minimising receive latency

In low-latency environments, a few microseconds can be gained by processing the application data directly in the receive thread, or synchronously with respect to the incoming network traffic, instead of queueing it for asynchronous processing by a delivery thread. This happens for data transmitted with the *max_latency* QoS setting at most a configurable value and the *transport_priority* QoS setting at least a configurable value. By default, these values are `inf` and the maximum transport priority, effectively enabling synchronous delivery for all data.

7.4.6 Maximum sample size

Eclipse Cyclone DDS provides a setting, `Internal/MaxSampleSize`, to control the maximum size of samples that the service is willing to process. The size is the size of the (CDR) serialised payload, and the limit holds both for built-in data and for application data. The (CDR) serialised payload is never larger than the in-memory representation of the data.

On the transmitting side, samples larger than `MaxSampleSize` are dropped with a warning in the. Eclipse Cyclone DDS behaves as if the sample never existed.

Similarly, on the receiving side, samples large than `MaxSampleSize` are dropped as early as possible, immediately following the reception of a sample or fragment of one, to prevent any resources from being claimed for longer than strictly necessary. Where the transmitting side completely ignores the sample, the receiving side pretends the sample has been correctly received and, at the acknowledges reception to the writer. This allows communication to continue.

When the receiving side drops a sample, readers will get a *sample lost* notification at the next sample that does get delivered to those readers. This condition means that again checking the info log is ultimately the only truly reliable way of determining whether samples have been dropped or not.

While dropping samples (or fragments thereof) as early as possible is beneficial from the point of view of reducing resource usage, it can make it hard to decide whether or not dropping a particular sample has been recorded in the log already. Under normal operational circumstances, only a single message will be recorded for each sample dropped, but it may on occasion report multiple events for the same sample.

Finally, it is technically allowed to set `MaxSampleSize` to very small sizes, even to the point that the discovery data can't be communicated anymore. The dropping of the discovery data will be duly reported, but the usefulness of such a configuration seems doubtful.

7.5 Network partition configuration

7.5.1 Network partition configuration overview

Network partitions introduce alternative multicast addresses for data. In the DDSI discovery protocol, a reader can override the default address at which it is reachable, and this feature of the discovery protocol is used to advertise alternative multicast addresses. The DDSI writers in the network will (also) multicast to such an alternative multicast address when multicasting samples or control data.

The mapping of a DCPS data reader to a network partition is indirect: first the DCPS partitions and topic are matched against a table of *partition mappings*, partition/topic combinations to obtain the name of a network partition, then the network partition name is used to find a addressing information.. This makes it easier to map many different partition/topic combinations to the same multicast address without having to specify the actual multicast address many times over.

If no match is found, the default multicast address is used.

7.5.2 Matching rules

Matching of a DCPS partition/topic combination proceeds in the order in which the partition mappings are specified in the configuration. The first matching mapping is the one that will be used. The `*` and `?` wildcards are available for the DCPS partition/topic combination in the partition mapping.

As mentioned earlier, Eclipse Cyclone DDS can be instructed to ignore all DCPS data readers and writers for certain DCPS partition/topic combinations through the use of *IgnoredPartitions*. The ignored partitions use the same matching rules as normal mappings, and take precedence over the normal mappings.

7.5.3 Multiple matching mappings

A single DCPS data reader can be associated with a set of partitions, and each partition/topic combination can potentially map to a different network partitions. In this case, the first matching network partition will be used. This does not affect what data the reader will receive; it only affects the addressing on the network.

7.6 Thread configuration

Eclipse Cyclone DDS creates a number of threads and each of these threads has a number of properties that can be controlled individually. The properties that can be controlled are:

- stack size,
- scheduling class, and
- scheduling priority.

The threads are named and the attribute `Threads/Thread[@name]` is used to set the properties by thread name. Any subset of threads can be given special properties; anything not specified explicitly is left at the default value.

The following threads exist:

- *gc*: garbage collector, which sleeps until garbage collection is requested for an entity, at which point it starts monitoring the state of Eclipse Cyclone DDS, pushing the entity through whatever state transitions are needed once it is safe to do so, ending with the freeing of the memory.
- *recv*: accepts incoming network packets from all sockets/ports, performs all protocol processing, queues (nearly) all protocol messages sent in response for handling by the timed-event thread, queues for delivery or, in special cases, delivers it directly to the data readers.
- *dq.builtins*: processes all discovery data coming in from the network.
- *lease*: performs internal liveliness monitoring of Eclipse Cyclone DDS.
- *tev*: timed-event handling, used for all kinds of things, such as: periodic transmission of participant discovery and liveliness messages, transmission of control messages for reliable writers and readers (except those that have their own timed-event thread), retransmitting of reliable data on request (except those that have their own timed-event thread), and handling of start-up mode to normal mode transition.

and, for each defined channel:

- *dq.channel-name*: deserialisation and asynchronous delivery of all user data.
- *tev.channel-name*: channel-specific 'timed-event' handling: transmission of control messages for reliable writers and readers and retransmission of data on request. Channel-specific threads exist only if the configuration includes an element for it or if an auxiliary bandwidth limit is set for the channel.

When no channels are explicitly defined, there is one channel named *user*.

7.7 Reporting and tracing

Eclipse Cyclone DDS can produce highly detailed traces of all traffic and internal activities. It enables individual categories of information, as well as having a simple verbosity level that enables fixed sets of categories.

The categorisation of tracing output is incomplete and hence most of the verbosity levels and categories are not of much use in the current release. This is an ongoing process and here we describe the target situation rather than the current situation.

All *fatal* and *error* messages are written both to the trace and to the `cyclonedds-error.log` file; similarly all 'warning' messages are written to the trace and the `cyclonedds-info.log` file.

The Tracing element has the following sub elements:

- *Verbosity*: selects a tracing level by enabled a pre-defined set of categories. The list below gives the known tracing levels, and the categories they enable:
  - *none*
  - *severe*: 'error' and 'fatal'
  - *warning*, *info*: severe + 'warning'
  - *config*: info + 'config'
  - *fine*: config + 'discovery'
  - *finer*: fine + 'traffic', 'timing' and 'info'
  - *finest*: fine + 'trace'
- *EnableCategory*: a comma-separated list of keywords, each keyword enabling individual categories. The following keywords are recognised:
  - *fatal*: all fatal errors, errors causing immediate termination
  - *error*: failures probably impacting correctness but not necessarily causing immediate termination.
  - *warning*: abnormal situations that will likely not impact correctness.
  - *config*: full dump of the configuration
  - *info*: general informational notices
  - *discovery*: all discovery activity
  - *data*: include data content of samples in traces
  - *timing*: periodic reporting of CPU loads per thread
  - *traffic*: periodic reporting of total outgoing data
  - *tcp*: connection and connection cache management for the TCP support
  - *throttle*: throttling events where the writer stalls because its WHC hit the high-water mark
  - *topic*: detailed information on topic interpretation (in particular topic keys)
  - *plist*: dumping of parameter lists encountered in discovery and inline QoS
  - *radmin*: receive buffer administration
  - *whc*: very detailed tracing of WHC content management

In addition, the keyword *trace* enables everything from *fatal* to *throttle*. The *topic* and *plist* ones are useful only for particular classes of discovery failures; and *radmin* and *whc* only help in analyzing the detailed behaviour of those two components and produce very large amounts of output.

- *OutputFile*: the file to write the trace to
- *AppendToFile*: boolean, set to `true` to append to the trace instead of replacing the file.

Currently, the useful verbosity settings are *config*, *fine* and *finest*.

Config writes the full configuration to the trace file as well as any warnings or errors, which can be a good way to verify everything is configured and behaving as expected.

Fine additionally includes full discovery information in the trace, but nothing related to application data or protocol activities. If a system has a stable topology, this will therefore typically result in a moderate size trace.

Finest provides a detailed trace of everything that occurs and is an indispensable source of information when analysing problems; however, it also requires a significant amount of time and results in huge log files.

Whether these logging levels are set using the verbosity level or by enabling the corresponding categories is immaterial.

7.8 Compatibility and conformance

7.8.1 Conformance modes

Eclipse Cyclone DDS operates in one of three modes: *pedantic*, *strict* and *lax*; the mode is configured using the `Compatibility/StandardsConformance` setting. The default is *lax*.

The first, *pedantic* mode, is of such limited utility that it will be removed.

The second mode, *strict*, attempts to follow the *intent* of the specification while staying close to the letter of it. Recent developments at the OMG have resolved these issues and this mode is no longer of any value.

The default mode, *lax*, attempts to work around (most of) the deviations of other implementations, and generally provides good interoperability without any further settings. In lax mode, the Eclipse Cyclone DDS not only accepts some invalid messages, it will even transmit them. The consequences for interoperability of not doing this are simply too severe. It should be noted that if one configures two Eclipse Cyclone DDS processes with different compliancy modes, the one in the stricter mode will complain about messages sent by the one in the less strict mode.

Compatibility issues with RTI

In *lax* mode, there should be no major issues with most topic types when working across a network, but within a single host there used to be an issue with the way RTI DDS uses, or attempts to use, its shared memory transport to communicate with peers even when they clearly advertises only UDP/IP addresses. The result is an inability to reliably establish bidirectional communication between the two.

Disposing data may also cause problems, as RTI DDS leaves out the serialised key value and instead expects the reader to rely on an embedded hash of the key value. In the strict modes, Eclipse Cyclone DDS requires a proper key value to be supplied; in the relaxed mode, it is willing to accept key hash, provided it is of a form that contains the key values in an unmangled form.

If an RTI DDS data writer disposes an instance with a key of which the serialised representation may be larger than 16 bytes, this problem is likely to occur. In practice, the most likely cause is using a key as string, either unbounded, or with a maximum length larger than 11 bytes. See the DDSI specification for details.

In *strict* mode, there is interoperation with RTI DDS, but at the cost of incredibly high CPU and network load, caused by a Heartbeats and AckNacks going back-and-forth between a reliable RTI DDS data writer and a reliable Eclipse Cyclone DDS data reader. The problem is that once Eclipse Cyclone DDS informs the RTI writer that it has received all data (using a valid AckNack message), the RTI writer immediately publishes a message listing the range of available sequence numbers and requesting an acknowledgement, which becomes an endless loop.

There is furthermore also a difference of interpretation of the meaning of the 'autodispose_unregistered_instances' QoS on the writer. Eclipse Cyclone DDS aligns with OpenSplice.

Compatibility issues with TwinOaks

Interoperability with TwinOaks CoreDX require (or used to require at some point in the past):

- `Compatibility/ManySocketsMode`: *true*

- `Compatibility/StandardsConformance`: *lax*
- `Compatibility/AckNackNumbitsEmptySet`: *0*
- `Compatibility/ExplicitlyPublishQosSetToDefault`: *true*

The `ManySocketsMode` option needed to be changed from the default, to ensure that each domain participant has a unique locator; this was needed because TwinOaks CoreDX DDS did not include the full GUID of a reader or writer if it needs to address just one, but this is probably no longer the case. Note that the (old) behaviour of TwinOaks CoreDX DDS has always been allowed by the specification.

The `Compatibility/ExplicitlyPublishQosSetToDefault` settings work around TwinOaks CoreDX DDS' use of incorrect default values for some of the QoS settings if they are not explicitly supplied during discovery. It may be that this is no longer the case.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index