

OpenSplice DDS

Version 5.x

C++ Reference Guide



OpenSplice DDS

C++ REFERENCE GUIDE



Part Number: OS-CPPREFG

Doc Issue 24, 29 March 2010

Copyright Notice

© 2010 PrismTech Limited. All rights reserved.

This document may be reproduced in whole but not in part.

The information contained in this document is subject to change without notice and is made available in good faith without liability on the part of PrismTech Limited or PrismTech Corporation.

All trademarks acknowledged.

A close-up, low-angle photograph of a computer keyboard, focusing on the central and right-hand keys. The keys are white with dark lettering. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

CONTENTS

Table of Contents

Preface

About the C++ Reference Guide	xvii
Contacts	xix

Introduction

About the C++ Reference Guide	3
Document Structure	3
Operations	4

API Reference

Chapter 1	DCPS API General Description	7
1.1	Thread Safety	8
1.2	Signal Handling	8
1.3	Memory Management	9
1.3.1	Reference Count	9
1.3.2	Reference Types	9
1.3.2.1	Pointer Types	9
1.3.2.2	Var Reference Types	10
1.3.2.3	Assignment	10
1.4	Listener Interfaces	12
1.5	Inheritance of Abstract Operations	13
Chapter 2	DCPS Modules	15
2.1	Functionality	15
2.2	Infrastructure Module	16
2.3	Domain Module	17
2.4	Topic-Definition Module	18
2.5	Publication Module	20
2.6	Subscription Module	21
Chapter 3	DCPS Classes and Operations	25
3.1	Infrastructure Module	26
3.1.1	Class Entity (abstract)	26
3.1.1.1	enable	27
3.1.1.2	get_instance_handle	29
3.1.1.3	get_listener (abstract)	30
3.1.1.4	get_qos (abstract)	30
3.1.1.5	get_status_changes	30

3.1.1.6	get_statuscondition	32
3.1.1.7	set_listener (abstract)	32
3.1.1.8	set_qos (abstract)	33
3.1.2	Class DomainEntity (abstract)	33
3.1.3	Struct QosPolicy	33
3.1.3.1	DeadlineQosPolicy	41
3.1.3.2	DestinationOrderQosPolicy	43
3.1.3.3	DurabilityQosPolicy	45
3.1.3.4	DurabilityServiceQosPolicy	48
3.1.3.5	EntityFactoryQosPolicy	51
3.1.3.6	GroupDataQosPolicy	51
3.1.3.7	HistoryQosPolicy	52
3.1.3.8	LatencyBudgetQosPolicy	55
3.1.3.9	LifespanQosPolicy	56
3.1.3.10	LivelinessQosPolicy	57
3.1.3.11	OwnershipQosPolicy	59
3.1.3.12	OwnershipStrengthQosPolicy	62
3.1.3.13	PartitionQosPolicy	62
3.1.3.14	PresentationQosPolicy	63
3.1.3.15	ReaderDataLifecycleQosPolicy	65
3.1.3.16	ReliabilityQosPolicy	66
3.1.3.17	ResourceLimitsQosPolicy	68
3.1.3.18	SchedulingQosPolicy	70
3.1.3.19	TimeBasedFilterQosPolicy	71
3.1.3.20	TopicDataQosPolicy	72
3.1.3.21	TransportPriorityQosPolicy	72
3.1.3.22	UserDataQosPolicy	73
3.1.3.23	WriterDataLifecycleQosPolicy	74
3.1.4	Listener Interface	75
3.1.5	Struct Status	76
3.1.5.1	InconsistentTopicStatus	80
3.1.5.2	LivelinessChangedStatus	80
3.1.5.3	LivelinessLostStatus	82
3.1.5.4	OfferedDeadlineMissedStatus	83
3.1.5.5	OfferedIncompatibleQosStatus	84
3.1.5.6	PublicationMatchedStatus	86
3.1.5.7	RequestedDeadlineMissedStatus	87
3.1.5.8	RequestedIncompatibleQosStatus	88
3.1.5.9	SampleLostStatus	90
3.1.5.10	SampleRejectedStatus	90
3.1.5.11	SubscriptionMatchedStatus	92
3.1.6	Class WaitSet	92

3.1.6.1	attach_condition	93
3.1.6.2	detach_condition	94
3.1.6.3	get_conditions	95
3.1.6.4	wait	96
3.1.7	Class Condition	98
3.1.7.1	get_trigger_value	99
3.1.8	Class GuardCondition	99
3.1.8.1	get_trigger_value (inherited)	100
3.1.8.2	set_trigger_value	100
3.1.9	Class StatusCondition	101
3.1.9.1	get_enabled_statuses	103
3.1.9.2	get_entity	104
3.1.9.3	get_trigger_value (inherited)	104
3.1.9.4	set_enabled_statuses	105
3.2	Domain Module	107
3.2.1	Class DomainParticipant	107
3.2.1.1	assert_liveliness	111
3.2.1.2	contains_entity	112
3.2.1.3	create_contentfilteredtopic	113
3.2.1.4	create_multitopic	114
3.2.1.5	create_publisher	116
3.2.1.6	create_subscriber	118
3.2.1.7	create_topic	121
3.2.1.8	delete_contained_entities	124
3.2.1.9	delete_contentfilteredtopic	125
3.2.1.10	delete_multitopic	126
3.2.1.11	delete_publisher	128
3.2.1.12	delete_subscriber	129
3.2.1.13	delete_topic	130
3.2.1.14	enable (inherited)	131
3.2.1.15	find_topic	132
3.2.1.16	get_builtin_subscriber	133
3.2.1.17	get_current_time	134
3.2.1.18	get_default_publisher_qos	135
3.2.1.19	get_default_subscriber_qos	136
3.2.1.20	get_default_topic_qos	137
3.2.1.21	get_discovered_participants	138
3.2.1.22	get_discovered_participant_data	139
3.2.1.23	get_discovered_topics	139
3.2.1.24	get_discovered_topic_data	139
3.2.1.25	get_domain_id	140
3.2.1.26	get_listener	140

3.2.1.27	get_qos	141
3.2.1.28	get_status_changes (inherited)	142
3.2.1.29	get_statuscondition (inherited)	142
3.2.1.30	ignore_participant	143
3.2.1.31	ignore_publication	143
3.2.1.32	ignore_subscription	143
3.2.1.33	ignore_topic	143
3.2.1.34	lookup_topicdescription	144
3.2.1.35	set_default_publisher_qos	145
3.2.1.36	set_default_subscriber_qos	146
3.2.1.37	set_default_topic_qos	147
3.2.1.38	set_listener	148
3.2.1.39	set_qos	151
3.2.2	Class DomainParticipantFactory	152
3.2.2.1	create_participant	153
3.2.2.2	delete_participant	156
3.2.2.3	get_default_participant_qos	157
3.2.2.4	get_instance	158
3.2.2.5	get_qos	159
3.2.2.6	lookup_participant	160
3.2.2.7	set_default_participant_qos	161
3.2.2.8	set_qos	162
3.2.2.9	delete_domain	163
3.2.2.10	lookup_domain	164
3.2.3	Class Domain	165
3.2.3.1	create_persistent_snapshot	165
3.2.4	DomainParticipantListener interface	167
3.2.4.1	on_data_available (inherited, abstract)	169
3.2.4.2	on_data_on_readers (inherited, abstract)	169
3.2.4.3	on_inconsistent_topic (inherited, abstract)	169
3.2.4.4	on_liveliness_changed (inherited, abstract)	170
3.2.4.5	on_liveliness_lost (inherited, abstract)	170
3.2.4.6	on_offered_deadline_missed (inherited, abstract)	170
3.2.4.7	on_offered_incompatible_qos (inherited, abstract)	170
3.2.4.8	on_publication_matched (inherited, abstract)	171
3.2.4.9	on_requested_deadline_missed (inherited, abstract)	171
3.2.4.10	on_requested_incompatible_qos (inherited, abstract)	171
3.2.4.11	on_sample_lost (inherited, abstract)	171
3.2.4.12	on_sample_rejected (inherited, abstract)	172
3.2.4.13	on_subscription_matched (inherited, abstract)	172
3.3	Topic-Definition Module	173
3.3.1	Class TopicDescription (abstract)	174

3.3.1.1	get_name	175
3.3.1.2	get_participant	175
3.3.1.3	get_type_name	176
3.3.2	Class Topic	177
3.3.2.1	enable (inherited)	178
3.3.2.2	get_inconsistent_topic_status	178
3.3.2.3	get_listener	179
3.3.2.4	get_name (inherited)	180
3.3.2.5	get_participant (inherited)	180
3.3.2.6	get_qos	180
3.3.2.7	get_status_changes (inherited)	181
3.3.2.8	get_statuscondition (inherited)	181
3.3.2.9	get_type_name (inherited)	182
3.3.2.10	set_listener	182
3.3.2.11	set_qos	184
3.3.3	Class ContentFilteredTopic	185
3.3.3.1	get_expression_parameters	186
3.3.3.2	get_filter_expression	187
3.3.3.3	get_name (inherited)	188
3.3.3.4	get_participant (inherited)	188
3.3.3.5	get_related_topic	188
3.3.3.6	get_type_name (inherited)	189
3.3.3.7	set_expression_parameters	189
3.3.4	Class MultiTopic	190
3.3.4.1	get_expression_parameters	191
3.3.4.2	get_name (inherited)	192
3.3.4.3	get_participant (inherited)	193
3.3.4.4	get_subscription_expression	193
3.3.4.5	get_type_name (inherited)	194
3.3.4.6	set_expression_parameters	194
3.3.5	TopicListener interface	195
3.3.5.1	on_inconsistent_topic (abstract)	196
3.3.6	Topic-Definition Type Specific Classes	197
3.3.6.1	Class TypeSupport (abstract)	197
3.3.6.2	get_type_name (abstract)	198
3.3.6.3	register_type (abstract)	198
3.3.6.4	Class FooTypeSupport	198
3.3.6.5	get_type_name	199
3.3.6.6	register_type	200
3.4	Publication Module	201
3.4.1	Class Publisher	202
3.4.1.1	begin_coherent_changes	204

3.4.1.2	copy_from_topic_qos	206
3.4.1.3	create_datawriter	207
3.4.1.4	delete_contained_entities	210
3.4.1.5	delete_datawriter	211
3.4.1.6	enable (inherited)	212
3.4.1.7	end_coherent_changes	212
3.4.1.8	get_default_datawriter_qos	213
3.4.1.9	get_listener	214
3.4.1.10	get_participant	215
3.4.1.11	get_qos	216
3.4.1.12	get_status_changes (inherited)	216
3.4.1.13	get_statuscondition (inherited)	217
3.4.1.14	lookup_datawriter	217
3.4.1.15	resume_publications	218
3.4.1.16	set_default_datawriter_qos	219
3.4.1.17	set_listener	220
3.4.1.18	set_qos	222
3.4.1.19	suspend_publications	223
3.4.1.20	wait_for_acknowledgments	225
3.4.2	Publication Type Specific Classes	226
3.4.2.1	Class DataWriter (abstract)	226
3.4.2.2	assert_liveliness	229
3.4.2.3	dispose (abstract)	230
3.4.2.4	dispose_w_timestamp (abstract)	231
3.4.2.5	enable (inherited)	231
3.4.2.6	get_key_value (abstract)	231
3.4.2.7	get_listener	232
3.4.2.8	get_liveliness_lost_status	232
3.4.2.9	get_matched_subscription_data	233
3.4.2.10	get_matched_subscriptions	234
3.4.2.11	get_offered_deadline_missed_status	234
3.4.2.12	get_offered_incompatible_qos_status	235
3.4.2.13	get_publication_matched_status	236
3.4.2.14	get_publisher	236
3.4.2.15	get_qos	237
3.4.2.16	get_status_changes (inherited)	238
3.4.2.17	get_statuscondition (inherited)	238
3.4.2.18	get_topic	238
3.4.2.19	lookup_instance (abstract)	239
3.4.2.20	register_instance (abstract)	239
3.4.2.21	register_instance_w_timestamp (abstract)	240
3.4.2.22	set_listener	240

3.4.2.23	set_qos	242
3.4.2.24	unregister_instance (abstract)	243
3.4.2.25	unregister_instance_w_timestamp (abstract)	244
3.4.2.26	wait_for_acknowledgments	244
3.4.2.27	write (abstract)	245
3.4.2.28	write_w_timestamp (abstract)	245
3.4.2.29	writediscard (abstract)	246
3.4.2.30	writediscard_w_timestamp (abstract)	246
3.4.2.31	Class FooDataWriter	246
3.4.2.32	assert_liveliness (inherited)	249
3.4.2.33	dispose	250
3.4.2.34	dispose_w_timestamp	253
3.4.2.35	enable (inherited)	255
3.4.2.36	get_key_value	255
3.4.2.37	get_listener (inherited)	256
3.4.2.38	get_liveliness_lost_status (inherited)	256
3.4.2.39	get_matched_subscription_data (inherited)	257
3.4.2.40	get_matched_subscriptions (inherited)	257
3.4.2.41	get_offered_deadline_missed_status (inherited)	257
3.4.2.42	get_offered_incompatible_qos_status (inherited)	257
3.4.2.43	get_publication_matched_status (inherited)	258
3.4.2.44	get_publisher (inherited)	258
3.4.2.45	get_qos (inherited)	258
3.4.2.46	get_status_changes (inherited)	258
3.4.2.47	get_statuscondition (inherited)	259
3.4.2.48	get_topic (inherited)	259
3.4.2.49	lookup_instance	259
3.4.2.50	register_instance	261
3.4.2.51	register_instance_w_timestamp	263
3.4.2.52	set_listener (inherited)	264
3.4.2.53	set_qos (inherited)	264
3.4.2.54	unregister_instance	264
3.4.2.55	unregister_instance_w_timestamp	268
3.4.2.56	write	269
3.4.2.57	write_w_timestamp	272
3.4.2.58	writediscard	273
3.4.2.59	writediscard_w_timestamp	277
3.4.3	PublisherListener Interface	279
3.4.3.1	on_liveliness_lost (inherited, abstract)	280
3.4.3.2	on_offered_deadline_missed (inherited, abstract)	280
3.4.3.3	on_offered_incompatible_qos (inherited, abstract)	280
3.4.3.4	on_publication_matched (inherited, abstract)	281

3.4.4	DataWriterListener Interface	281
3.4.4.1	on_liveliness_lost (abstract)	282
3.4.4.2	on_offered_deadline_missed (abstract)	283
3.4.4.3	on_offered_incompatible_qos (abstract)	284
3.4.4.4	on_publication_matched (abstract).	285
3.5	Subscription Module	286
3.5.1	Class Subscriber	287
3.5.1.1	begin_access	289
3.5.1.2	copy_from_topic_qos	289
3.5.1.3	create_datareader	291
3.5.1.4	delete_contained_entities	294
3.5.1.5	delete_datareader	295
3.5.1.6	enable (inherited)	296
3.5.1.7	end_access	296
3.5.1.8	get_datareaders	296
3.5.1.9	get_default_datareader_qos	297
3.5.1.10	get_listener	298
3.5.1.11	get_participant	299
3.5.1.12	get_qos	299
3.5.1.13	get_status_changes (inherited)	300
3.5.1.14	get_statuscondition (inherited).	300
3.5.1.15	lookup_datareader	301
3.5.1.16	notify_datareaders	301
3.5.1.17	set_default_datareader_qos	303
3.5.1.18	set_listener	304
3.5.1.19	set_qos	306
3.5.2	Subscription Type Specific Classes.	308
3.5.2.1	Class DataReader (abstract)	308
3.5.2.2	create_querycondition	313
3.5.2.3	create_readcondition.	314
3.5.2.4	delete_contained_entities	315
3.5.2.5	delete_readcondition.	317
3.5.2.6	enable (inherited)	318
3.5.2.7	get_key_value (abstract).	318
3.5.2.8	get_listener	318
3.5.2.9	get_liveliness_changed_status	319
3.5.2.10	get_matched_publication_data	320
3.5.2.11	get_matched_publications	320
3.5.2.12	get_qos	320
3.5.2.13	get_requested_deadline_missed_status	321
3.5.2.14	get_requested_incompatible_qos_status	322
3.5.2.15	get_sample_lost_status	323

3.5.2.16	get_sample_rejected_status	325
3.5.2.17	get_status_changes (inherited)	326
3.5.2.18	get_statuscondition (inherited)	326
3.5.2.19	get_subscriber	326
3.5.2.20	get_subscription_matched_status	327
3.5.2.21	get_topicdescription	327
3.5.2.22	lookup_instance (abstract)	328
3.5.2.23	read (abstract)	328
3.5.2.24	read_instance (abstract)	328
3.5.2.25	read_next_instance (abstract)	329
3.5.2.26	read_next_instance_w_condition (abstract)	329
3.5.2.27	read_next_sample (abstract)	330
3.5.2.28	read_w_condition (abstract)	330
3.5.2.29	return_loan (abstract)	330
3.5.2.30	set_listener	331
3.5.2.31	set_qos	333
3.5.2.32	take (abstract)	334
3.5.2.33	take_instance (abstract)	335
3.5.2.34	take_next_instance (abstract)	335
3.5.2.35	take_next_instance_w_condition (abstract)	336
3.5.2.36	take_next_sample (abstract)	336
3.5.2.37	take_w_condition (abstract)	336
3.5.2.38	wait_for_historical_data	337
3.5.2.39	Class FooDataReader	338
3.5.2.40	create_querycondition (inherited)	343
3.5.2.41	create_readcondition (inherited)	343
3.5.2.42	delete_contained_entities (inherited)	344
3.5.2.43	delete_readcondition (inherited)	344
3.5.2.44	enable (inherited)	344
3.5.2.45	get_key_value	344
3.5.2.46	get_listener (inherited)	345
3.5.2.47	get_liveliness_changed_status (inherited)	345
3.5.2.48	get_matched_publication_data (inherited)	345
3.5.2.49	get_matched_publications (inherited)	345
3.5.2.50	get_qos (inherited)	346
3.5.2.51	get_requested_deadline_missed_status (inherited)	346
3.5.2.52	get_requested_incompatible_qos_status (inherited)	346
3.5.2.53	get_sample_lost_status (inherited)	346
3.5.2.54	get_sample_rejected_status (inherited)	346
3.5.2.55	get_status_changes (inherited)	347
3.5.2.56	get_statuscondition (inherited)	347
3.5.2.57	get_subscriber (inherited)	347

3.5.2.58	get_subscription_matched_status (inherited)	347
3.5.2.59	get_topicdescription (inherited)	348
3.5.2.60	lookup_instance	348
3.5.2.61	read	349
3.5.2.62	read_instance	353
3.5.2.63	read_next_instance	355
3.5.2.64	read_next_instance_w_condition	357
3.5.2.65	read_next_sample	359
3.5.2.66	read_w_condition	359
3.5.2.67	return_loan	361
3.5.2.68	set_listener (inherited)	363
3.5.2.69	set_qos (inherited)	363
3.5.2.70	take	364
3.5.2.71	take_instance	365
3.5.2.72	take_next_instance	367
3.5.2.73	take_next_instance_w_condition	369
3.5.2.74	take_next_sample	370
3.5.2.75	take_w_condition	371
3.5.2.76	wait_for_historical_data (inherited)	372
3.5.3	Class DataSample	373
3.5.4	Struct SampleInfo	373
3.5.4.1	SampleInfo	373
3.5.5	SubscriberListener Interface	376
3.5.5.1	on_data_available (inherited, abstract)	378
3.5.5.2	on_data_on_readers (abstract)	378
3.5.5.3	on_liveliness_changed (inherited, abstract)	379
3.5.5.4	on_requested_deadline_missed (inherited, abstract)	380
3.5.5.5	on_requested_incompatible_qos (inherited, abstract)	380
3.5.5.6	on_sample_lost (inherited, abstract)	380
3.5.5.7	on_sample_rejected (inherited, abstract)	380
3.5.5.8	on_subscription_matched (inherited, abstract)	381
3.5.6	DataReaderListener Interface	381
3.5.6.1	on_data_available (abstract)	382
3.5.6.2	on_liveliness_changed (abstract)	383
3.5.6.3	on_requested_deadline_missed (abstract)	384
3.5.6.4	on_requested_incompatible_qos (abstract)	385
3.5.6.5	on_sample_lost (abstract)	386
3.5.6.6	on_sample_rejected (abstract)	387
3.5.6.7	on_subscription_matched (abstract)	388
3.5.7	Class ReadCondition	388
3.5.7.1	get_datareader	389
3.5.7.2	get_instance_state_mask	390

3.5.7.3	get_sample_state_mask	390
3.5.7.4	get_trigger_value (inherited)	391
3.5.7.5	get_view_state_mask	391
3.5.8	Class QueryCondition	392
3.5.8.1	get_datareader (inherited)	393
3.5.8.2	get_instance_state_mask (inherited)	394
3.5.8.3	get_query_parameters	394
3.5.8.4	get_query_expression	395
3.5.8.5	get_sample_state_mask (inherited)	396
3.5.8.6	get_trigger_value (inherited)	396
3.5.8.7	get_view_state_mask (inherited)	396
3.5.8.8	set_query_parameters	396
Appendix A	Quality Of Service	401
	Affected Entities	401
	Basic Usage	401
	DataReaderQos	403
	DataWriterQos	405
	DomainParticipantFactoryQos	408
	DomainParticipantQos	409
	PublisherQos	410
	SubscriberQos	412
	TopicQos	413
		415
Appendix B	API Constants and Types	417
Appendix C	Platform Specific Model IDL Interface	423
	dds_dcps.idl	423
Appendix D	SampleStates, ViewStates and InstanceStates	451
	SampleInfo Class	451
	sample_state	451
	instance_state	452
	view_state	454
	State Masks	456
	Operations Concerning States	457
Appendix E	Class Inheritance	461
Appendix F	Listeners, Conditions and Waitsets	463
	Communication Status Event	464
	Listeners	467

Conditions and Waitsets	469
StatusCondition Trigger State	472
ReadCondition and QueryCondition Trigger State	472
GuardCondition Trigger State	473
Appendix G Topic Definitions	475
Topic Definition Example	475
Complex Topics.	476
IDL Pre-processor	476
IDL-to-Host Language Mapping	476
Data Distribution Service IDL Keywords	476
Data Distribution Service IDL Pragma Keylist.	477
Data Distribution Service IDL subset in BNF-notation	477
Appendix H DCPS Queries and Filters	481
SQL Grammar in BNF	481
SQL Token Expression	482
SQL Examples.	483
Bibliography	487
Glossary	491
Index	495

Preface

About the C++ Reference Guide

The *C++ Reference Guide* provides a detailed explanation of the OpenSplice DDS (*Subscription Paradigm for the Logical Interconnection of Concurrent Engines*) Application Programming Interfaces for the C++ language.

This reference guide is based on the OMG's *Data Distribution Service Specification* and *C++ Language Mapping Specification*.

The C++ Reference Guide focuses on the Data Centric Publish Subscribe (DCPS) layer and does not cover the DLRL layer. The purpose of the DCPS is the distribution of data (publish/subscribe). The structure of the DCPS is divided into five modules. Each module consists of several classes, which in turn generally contain several operations.

Intended Audience

The *C++ Reference Guide* is intended to be used by C++ programmers who are using OpenSplice DDS to develop applications.

Organisation

The C++ Reference Guide is organised as follows:

An *Introduction* describes the details of the document structure.

A *Bibliography* and *Glossary* are provided at the end of the guide with, respectively, details of recommended references and definitions.

Chapter 1, *DCPS API General Description*, is a general description of the DCPS API and its error codes.

Chapter 2, *DCPS Modules*, provides the detailed description of the DCPS modules.

Chapter 3, *DCPS Classes and Operations*, provides the detailed description of the DCPS classes, structs and operations.

The following appendices are included, as well as a *Bibliography* containing references material and *Glossary*:

Appendix A, *Quality Of Service*

Appendix B, *API Constants and Types*

Appendix C, *Platform Specific Model IDL Interface*

Appendix D, *SampleStates, ViewStates and InstanceStates*

Appendix E, *Class Inheritance*

Appendix F, *Listeners, Conditions and Waitsets*

Appendix G, *Topic Definitions*

Appendix H, *DCPS Queries and Filters*

Conventions

The conventions listed below are used to guide and assist the reader in understanding the C++ Reference Guide.



Item of special significance or where caution needs to be taken.

i

Item contains helpful hint or special information.

WIN

Information applies to Windows (e.g. NT, 2000, XP) only.

UNIX

Information applies to Unix based systems (e.g. Solaris) only.

C

C language specific

C++

C++ language specific

Java

Java language specific

Hypertext links are shown as [*blue italic underlined*](#).

On-Line (PDF) versions of this document: Items shown as cross references, e.g. *Contacts* on page xix, are as hypertext links: click on the reference to go to the item.

```
% Commands or input which the user enters on the
   command line of their computer terminal
```

Courier fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to "example" and kind field to an empty string
newName[0] = new NameComponent ("example", "");
```

Italics and ***Italic Bold*** are used to indicate new terms, or emphasise an item.

Arial Bold is used to indicate user related actions, e.g. **File | Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900

Fax: +44 (0)191 497 9901

Web:

<http://www.prismtech.com>

Technical questions:

crc@prismtech.com (Customer Response Center)

Sales enquiries:

sales@prismtech.com

A close-up, low-angle shot of a computer keyboard, focusing on the keys. The image is overlaid with a white grid pattern, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

INTRODUCTION

About the C++ Reference Guide

Document Structure

The C++ Reference Guide document structure is based on the structure of the DCPS Platform Independent Model (DCPS PIM) of the Data Distribution Service Specification. The detailed description is subdivided into the PIM Modules, which are then subdivided into classes.

Some of the classes are implemented as structs in the DCPS Platform Specific Model (DCPS PSM) of the Data Distribution Service Specification, as indicated in the Interface Description Language (IDL) chapter of the PSM (see Appendix C, *Platform Specific Model IDL Interface*). These structs are described in the respective chapters.

- In the classes as described in the PIM, which are implemented as a class in the PSM, the operations are described in detail.
- In the classes as described in the PIM, which are implemented as a struct in the PSM, the struct contents are described in detail.
- The order of the modules and classes is conform the PIM part.
- The order of the operations or struct contents is alphabetical.
- Each description of a class or struct starts with the API description header file.

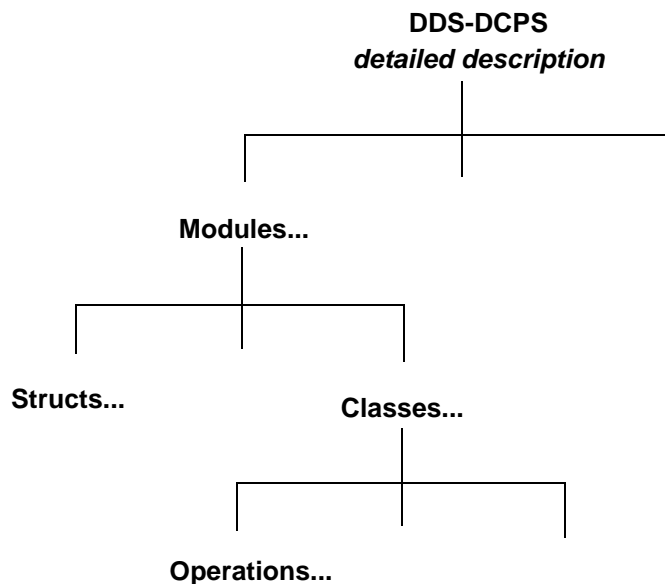


Figure 1 C++ Reference Guide Document Structure

Operations

Several types of operations are described in this manual. The different types of operations are: basic, inherited, abstract and abstract interface. All operations of any type can be found in their respective class. The details of their description depends on the type of operation.

Basic operations are described in detail in the class they are implemented in.

- Inherited operations only refer to the operation in the class they are inherited from. The detailed description is not repeated.
- Abstract operations only refer to the type specific implementations in their respective derived class. The detailed description is not repeated.
- Abstract operations which are implemented as an interface (`Listeners`), are described in detail in their class. These operations must be implemented in the application.

A close-up, low-angle shot of a computer keyboard, focusing on the central and right-hand keys. The keys are white with dark lettering. A semi-transparent grid of thin white lines is overlaid on the entire image, creating a technical or digital aesthetic. The lighting is soft, highlighting the texture of the keys and the grid pattern.

API REFERENCE


CHAPTER

1 DCPS API General Description

The structure of the DCPS is divided into modules, which are described in detail in the next chapter. Each module consists of several classes, which in turn may contain several operations.

Some of these operations have an operation return code of type `ReturnCode_t`, which is defined in the next table:

Table 1 Return Codes

ReturnCode_t	Return Code Description
RETCODE_OK	Successful return
RETCODE_ERROR	Generic, unspecified error
RETCODE_BAD_PARAMETER	Illegal parameter value
RETCODE_UNSUPPORTED	Unsupported operation or <code>QoSPolicy</code> setting. Can only be returned by operations that are optional or operations that uses an optional <code><Entity>QoS</code> as a parameter
RETCODE_ALREADY_DELETED	The object target of this operation has already been deleted
RETCODE_OUT_OF_RESOURCES	Service ran out of the resources needed to complete the operation
RETCODE_NOT_ENABLED	Operation invoked on an <code>Entity</code> that is not yet enabled
RETCODE_IMMUTABLE_POLICY	Application attempted to modify an immutable <code>QoSPolicy</code>
RETCODE_INCONSISTENT_POLICY	Application specified a set of policies that are not consistent with each other
RETCODE_PRECONDITION_NOT_MET	A pre-condition for the operation was not met
RETCODE_TIMEOUT	The operation timed out
RETCODE_ILLEGAL_OPERATION	An operation was invoked on an inappropriate object or at an inappropriate time (as determined by <code>QoSPolicies</code> that control the behaviour of the object in question). There is no precondition that could be changed to make the operation succeed.  This code can not be returned in C++.
RETCODE_NO_DATA	Indicates a situation where the operation did not return any data

The name scope (name space) of these return codes is DDS. The operation return codes `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED` and `RETCODE_ALREADY_DELETED` are default for operations that return an operation return code and are therefore not explicitly mentioned in the DDS specification. However, in this manual they are mentioned along with each operation.

Some operations are not implemented. These operations are mentioned including their synopsis, but not described in this manual and return `RETCODE_UNSUPPORTED` when called from the application. See Appendix B (API Constants and Types).

The return code `RETCODE_ILLEGAL_OPERATION` can never be returned in C++: it indicates that you tried to invoke an operation on the wrong class, which in a real Object Oriented language like C++ is never possible.

1.1 Thread Safety

All operations are thread safe.

1.2 Signal Handling



The Data Distribution Service sets signal handlers in order to assure that resources are released when signals that terminate the application process are cached. These signal handlers only call the exit function in order to force exit handlers to be activated.

If the application needs to set signal handlers for its own use, two situations can occur. In the first case the application sets a signal handler for a specific signal while the Data Distribution Service has not set a handler yet. The Data Distribution Service will not set its own handler in this case, but expects the application signal handler to call the exit function when the signal is meant to terminate the process. In the second case the Data Distribution Service has already set a signal handler for a specific signal and the application program redefines the signal handling by setting its own handler. In that case the application should either chain the Data Distribution Service signal handler (to be executed as last) or to call the exit function itself when the cached signal is meant to terminate the application process.

The Data Distribution Service will conditionally set the signal handlers when creating the `DomainParticipantFactory`, which is the first call to `DDS::DomainParticipantFactory::get_instance` for C++.

The Data Distribution Service only sets signal handlers for signals that have the default behaviour of terminating the process without dumping a core.

1.3 Memory Management

When objects are being created, they will occupy memory space. To avoid memory leaks when they are not used any more, these objects will have to be deleted in order to release the memory space. However, when using pointers, it is difficult to keep track of which object has been released and which has not. When objects are not being released, the memory leak will finally use up all the resources and the application will fail.

1.3.1 Reference Count

The DDS API is described as a collection of IDL interfaces in the PSM. According to the IDL to C++ language mapping these interfaces must be mapped onto C++ classes that inherit from a `CORBA::Object` class. OpenSplice can currently *borrow* this class from any ORB installed on your system, but it also provides its own implementation libraries: which library is used depends on whether you use the Corba C++ API (CCPP) or the standalone C++ API (SACPP).

In order to cope with the memory management problems described above, CORBA objects keep some internal administration. In this administration, a reference count is included. This reference count holds the number of references to the object (assuming ownership). In other words, when a second reference is being made to the same object, the reference count in the internal administration of the object, must be increased. This way, both references may assume ownership of the same object. When one of the references runs out of scope, the reference count must be decreased by one. In this case the object must not be released because the reference count has not reach zero yet. Only when the second reference runs out of scope, the reference count reaches zero and the object must be released.

1.3.2 Reference Types

CORBA defines two types of references. The first one is the basic `<class>_ptr` type. When this type is used, the application must explicitly increase or decrease the reference count. The second one is the `<class>_var` type. This type is a smart pointer, which automatically updates the reference count of an object when that object is assigned to it and also updates the reference of the previous assigned object. When this type is used, the application does not have to increase or decrease the reference count. Best practice is to use these `<class>_var` types instead of the `<class>_ptr` types. However, under certain conditions a `<class>_ptr` type must be used (refer to Section 1.3.2.2, *Var Reference Types*).

1.3.2.1 Pointer Types

When using `<class>_ptr` types the application must explicitly increase or decrease the reference count, by using the CORBA defined functions:

- *_duplicate* - creates another reference to the object. The object is not being copied but only the reference count in the internal administration of the object is increased and a new `<class>_ptr` type is returned. Both references have ownership. In other words, when one of them runs out of scope, the reference count must be decreased by calling *_release*. Only when *_release* is called for both of them, the object is removed;
- *_release* - informs the CORBA object that the application will not be using the reference any more. As a result, the operation will explicitly decrease the reference count of an object. After releasing, the application must not use the reference because from this moment on, it is unknown whether the object still exists.

When more references to `<class>_ptr` type are made by assignment, the reference count is not increased. When more references to `<class>_ptr` type are made by *_duplicate*, the reference count is increased on every call. Therefore *_release* must be called once for every *_duplicate* to decrease the reference count.

1.3.2.2 Var Reference Types

To prevent errors, CORBA defines the `<class>_var` types which assumes ownership of the object it is referring to. An `_var` type is considered to be a smart pointer, which not only includes the reference to the object but also automatically updates the internal reference count of the object.

1.3.2.3 Assignment

Assignment for `<class>_ptr` types and `<class>_var` types is defined for:

```
<class>_ptr types to <class>_var types
<class>_var types to <class>_var types
<class>_var types to <class>_ptr types
```

For instance, the result of a `create_publisher` (which returns a `Publisher_ptr`) can directly be assigned to a `Publisher_var` type. This assignment would transfer ownership of the `Publisher` object to the reference of `Publisher_var` type.

```
My_Publisher_var = create_publisher(PUBLISHER_QOS_DEFAULT,
    PublisherListener::_nil());
```

This assignment will wrap the return type `Publisher_ptr` in type `Publisher_var` and transfer ownership to `My_Publisher_var`. In other words, when `My_Publisher_var` runs out of scope, the `Publisher` is automatically removed.

The next assignment does not concern `<class>_var` types, but is only presented to show what will happen when a `<class>_var` type is not used.

```
My_Publisher_ptr = create_publisher(PUBLISHER_QOS_DEFAULT,
    PublisherListener::_nil());
```


This assignment will not wrap, but only makes a copy of the return type `Publisher_ptr`. The reference count is not automatically updated and the application has to make sure to release the object. In other words, when `My_Publisher_ptr` runs out of scope, the `Publisher` is not automatically removed and can not be removed any more since there is no reference available (not even by `delete_contained_entities`).

```
Another_Publisher_var = My_Publisher_var;
```

This assignment will create another reference to the `Publisher`. The object is not being copied but only the reference count in the internal administration of the `Publisher_var` type is increased. Both `Another_Publisher_var` and `My_Publisher_var` have ownership. In other words, when one of them runs out of scope, the reference count is decreased. Only when both of them run out of scope, the `Publisher` is removed.

```
Another_Publisher_ptr = My_Publisher_var;
```

This assignment will type cast the type `Publisher_var` to type `Publisher_ptr` and only makes a copy of the reference `My_Publisher_var`. The ownership is not transferred and the application may not release the object on account of `Another_Publisher_ptr` because the internal reference count was not increased. However, the application must be careful not to use `Another_Publisher_ptr` when `My_Publisher_var` runs out of scope because in that case, the `Publisher` is automatically removed and the `Another_Publisher_ptr` variable is invalid.

Var Reference Types Side Effect

As mentioned, under certain conditions a `<class>_ptr` type must be used, because the `<class>_var` type will cause a problem when a type cast is being done on an object. For example, the cast of an object of class `DataWriter` to the class `<type>DataWriter`.

When creating a `DataWriter`, the `create_datawriter` operation returns a generic `DataWriter_ptr` type (which we assign to `DataWriter_var`). However, an object of the `DataWriter` class does not have a `write` operation. To be able to use such a typed operation, the application must perform a dynamic cast to the `<type>DataWriter_ptr` type. For example, when we have a `DataWriter` for the type `Foo`, it looks like:

```
GenericWriter_var = create_datawriter(MyTopic,
    MyWriterQos, _nil);
My_Writer_var = dynamic_cast<FooDataWriter_ptr>
    GenericWriter_var.in());
```

The problem here is that there are two `DataWriter_var` type references to the object without a duplication. This is because a cast only copies the same information and does not increase the internal counter. In other words, both `DataWriter_var`

types considers to be the sole owner of the object, and therefore do not increase its reference counter. When one of the `DataWriter_var` types run out of scope, the entire object is removed, because the internal administration only had one reference accounted for. Therefore, when the second `DataWriter_var` type runs out of scope, the behaviour of your application has become undefined (most probably the application will eventually crash). This problem can be solved by using an explicit call to the `DataWriter::_duplicate` operation before assigning it to the second `_var` type, or by using a `DataWriter_ptr` type instead of a `DataWriter_var` type because a `DataWriter_ptr` type does not automatically decrease the reference counter when it runs out of scope.

```
GenericWriter_ptr = create_datawriter(MyTopic,
    MyWriterQos, _nil);

My_Writer_var =
    dynamic_cast<FooDataWriter_ptr>(GenericWriter_ptr.in());
```

Note in this case that `GenericWriter_ptr` must not be used after the assignment, since it will not be valid as soon as `My_Writer_var` runs out of scope.

1.4 Listener Interfaces

The `Listener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous communication status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc.

The `Listener` interfaces are designed as an interface at PIM level. In other words, such an interface is part of the application which must implement the interface operations. A user defined class for these operations must be provided by the application which must extend from the *specific* `Listener` class (according to the IDL-to-C++ specification an interface in IDL is mapped on a class in the C++ programming language). *All* `Listener` operations *must* be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

Each DCPS `Entity` supports its own specialized kind of `Listener`. Therefore, the following `Listeners` are available:

- `DomainParticipantListener`
- `TopicListener`
- `PublisherListener`
- `DataWriterListener`
- `SubscriberListener`
- `DataReaderListener`

Since a `DataReader` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` must be of type `DataReaderListener`. A user defined class must be provided by the application (for instance `My_DataReaderListener`) which must extend from the `DataReaderListener` class. *All* `DataReaderListener` operations *must* be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

As an example, one of the operations in the `DataReaderListener` is the `on_liveliness_changed`. This operation (implemented by the application) will be called by the Data Distribution Service when the liveliness of the associated `DataWriter` has changed. In other words, it serves as a callback function to the event of a change in liveliness. The parameters of the operation are supplied by the Data Distribution Service. In this example, the reference to the `DataReader` and the status of the liveliness are provided.

1.5 Inheritance of Abstract Operations

The information provided in this guide is based on:

- the PIM part of the DDS-DCPS specification for module descriptions
- the PSM part of the DDS-DCPS specification for class and operation descriptions.

Refer to the OMG's *Data Distribution Service Revised Final Adopted Specification, ptc/04-03-07*, for additional information.

At PIM level, inheritance is used to define abstract classes and operations. The OMG IDL PSM defines the interface for an application to interact with the Data Distribution Service (see Appendix C (Platform Specific Model IDL Interface)). The DCPS API for the C++ language is as specified in the OMG's *C++ Language Mapping Specification*.

Inheritance of operations is not implemented when different type parameters for the same operation are used. In this case operations are implemented in their respective derived class (e.g. `get_qos` and `set_qos`). These operations are commented out in the IDL PSM.

CHAPTER

2 DCPS Modules

DCPS is divided into five modules, which are described briefly in this chapter. Each module consists of several classes as defined at PIM level in the DDS-DCPS specification. Some of the classes as described in the PIM are implemented as a struct in the PSM; these classes are treated as a class in this chapter according to the PIM with a remark about their implementation (struct). In the next chapter their actual implementations are described.

Each class contains several operations, which may be abstract. Those classes, which are implemented as a struct do not have any operations. The modules and the classes are ordered conform the DDS-DCPS specification. The classes, interfaces, structs and operations are described in the next chapter.

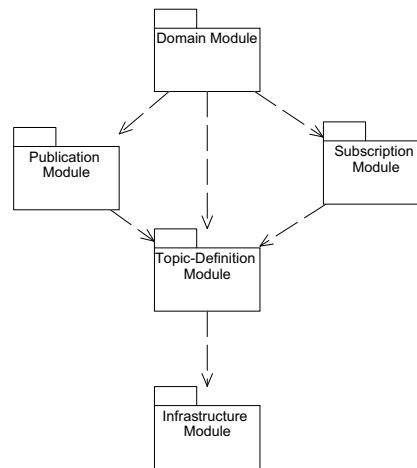


Figure 2 DCPS Module Composition

2.1 Functionality

The modules listed below provide the associated functions in the Data Distribution Service:

Infrastructure Module - This module defines the abstract classes and interfaces, which are refined by the other modules. It also provides the support for the interaction between the application and the Data Distribution Service (event-based and state-based);

Domain Module - This module contains the `DomainParticipant` class, which is the entry point of the application and `DomainParticipantListener` interface;

Topic-Definition Module - This module contains the `Topic`, `ContentFilteredTopic` and `MultiTopic` classes. It also contains the `TopicListener` interface and all support to define `Topic` objects and assign `QosPolicy` settings to them;

Publication Module - This module contains the `Publisher` and `DataWriter` classes. It also contains the `PublisherListener` and `DataWriterListener` interfaces;

Subscription Module - This module contains the `Subscriber`, `DataReader`, `ReadCondition` and `QueryCondition` classes. It also contains the `SubscriberListener` and `DataReaderListener` interfaces.

2.2 Infrastructure Module

This module defines the abstract classes and interfaces, which, in the PIM definition, are refined by the other modules. It also provides the support for the interaction between the application and the Data Distribution Service (event-based and state-based). The event-based interaction is supported by `Listeners`, the state-based interaction is supported by `WaitSets` and `Conditions`.

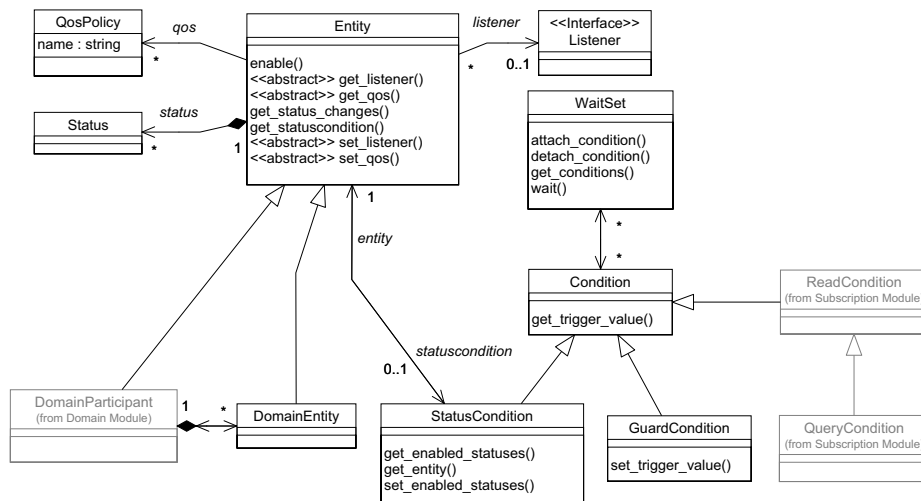


Figure 3 DCPS Infrastructure Module's Class Model

This module contains the following classes:

- `Entity` (abstract)
- `DomainEntity` (abstract)

- QosPolicy (abstract, struct)
- Listener (interface)
- Status (abstract, struct)
- WaitSet
- Condition
- GuardCondition
- StatusCondition

2.3 Domain Module

This module contains the class `DomainParticipant`, which acts as an entry point of the Data Distribution Service and acts as a factory for many of the classes. The `DomainParticipant` also acts as a container for the other objects that make up the Data Distribution Service. It isolates applications within the same `Domain` from other applications in a different `Domain` on the same set of computers. A `Domain` is a “virtual network” and applications with the same `domainId` are isolated from applications with a different `domainId`. In this way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.

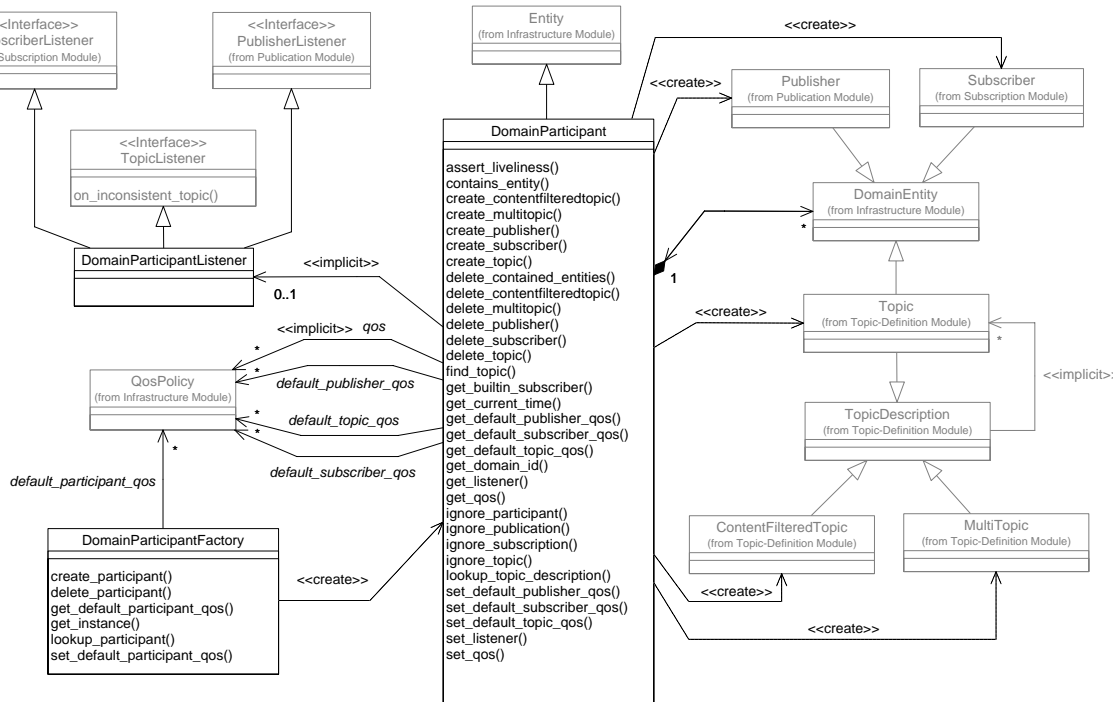


Figure 4 DCPS Domain Module's Class Model

This module contains the following classes:

- DomainParticipant
- DomainParticipantFactory
- DomainParticipantListener (interface)
- Domain (*not depicted*)

2.4 Topic-Definition Module

This module contains the Topic, ContentFilteredTopic and MultiTopic classes. It also contains the TopicListener interface and all support to define Topic objects and assign QoSPolicy settings to them.

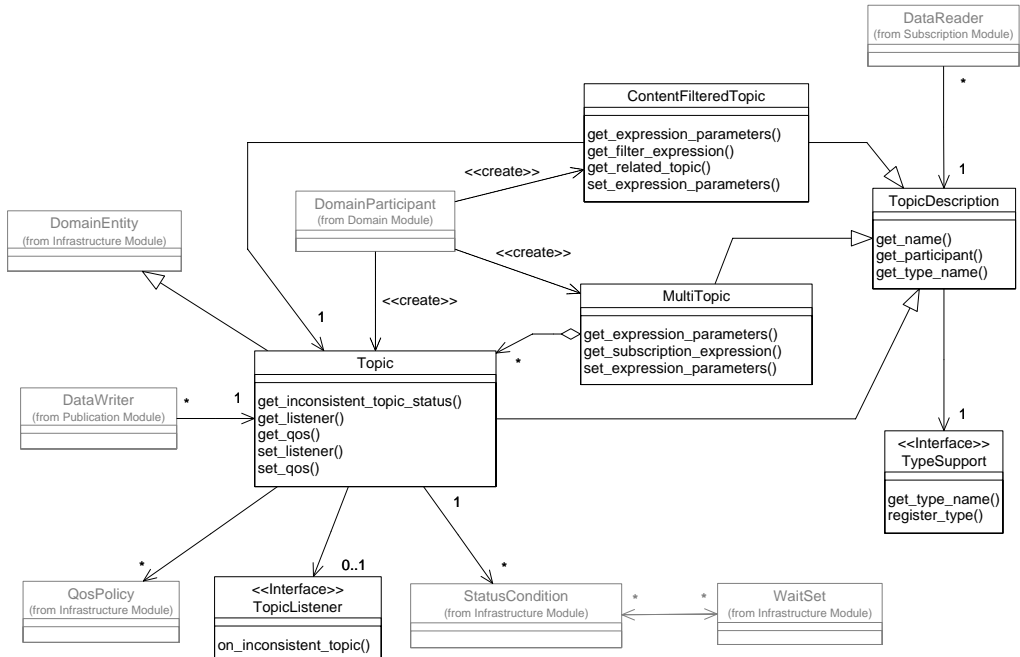


Figure 5 DCPS Topic-Definition Module's Class Model

This module contains the following classes:

- TopicDescription (abstract)
- Topic
- ContentFilteredTopic
- MultiTopic
- TopicListener (interface)
- Topic-Definition type specific classes

“Topic-Definition type specific classes” contains the generic class and the generated data type specific classes. In case of data type `Foo` (this also applies to other types); “Topic-Definition type specific classes” contains the following classes:

- TypeSupport (abstract)
- FooTypeSupport

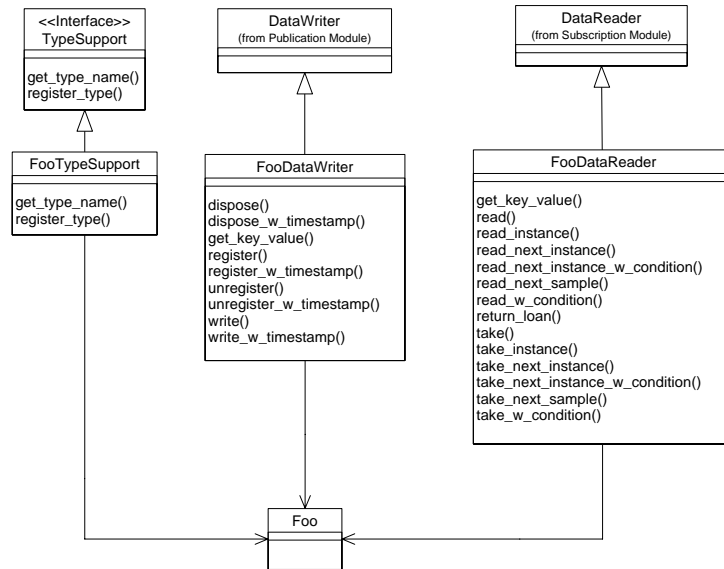


Figure 6 Data Type “Foo” Typed Classes for Pre-processor Generation

2.5 Publication Module

This module supports writing of the data, it contains the `Publisher` and `DataWriter` classes. It also contains the `PublisherListener` and `DataWriterListener` interfaces. Furthermore, it contains all support needed for publication.

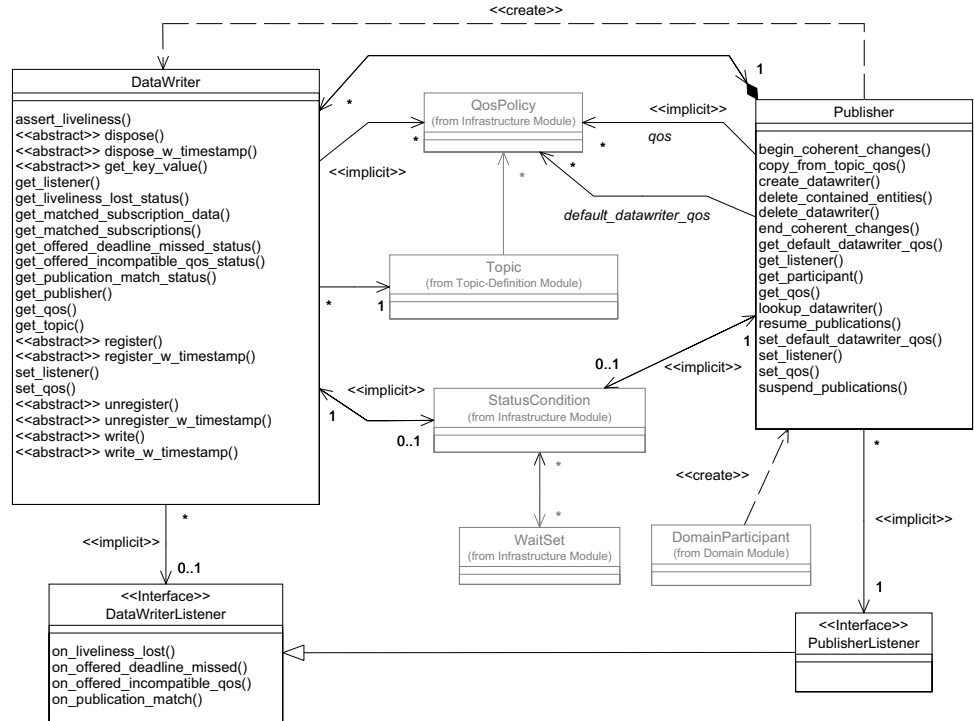


Figure 7 DCPS Publication Module's Class Model

This module contains the following classes:

- Publisher
- Publication type specific classes
- PublisherListener (interface)
- DataWriterListener (interface)

“Publication type specific classes” contains the generic class and the generated data type specific classes. In case of data type Foo (this also applies to other types); “Publication type specific classes” contains the following classes:

- DataWriter (abstract)
- FooDataWriter

2.6 Subscription Module

This module supports access to the data, it contains the Subscriber, DataReader, ReadCondition and QueryCondition classes. It also contains the SubscriberListener and DataReaderListener interfaces. Furthermore, it contains all support needed for subscription.

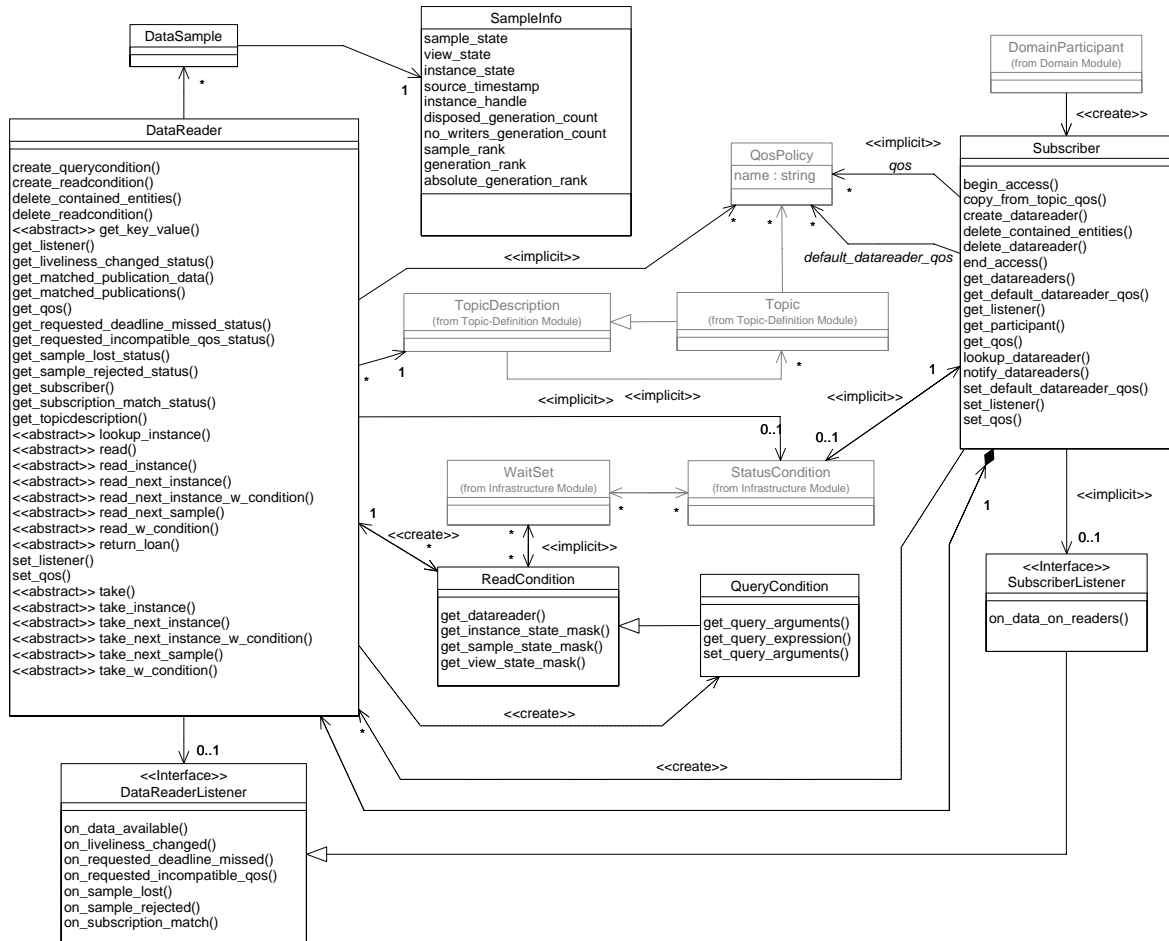


Figure 8 DCPS Subscription Module's Class Model

This module contains the following classes:

- Subscriber
- **Subscription type specific classes**
- `DataSample`
- `SampleInfo` (struct)
- `SubscriberListener` (interface)
- `DataReaderListener` (interface)
- `ReadCondition`
- `QueryCondition`

“Subscription type specific classes” contains the generic class and the generated data type specific classes. In case of data type `Foo` (this also applies to other types); “Subscription type specific classes” contains the following classes:

- `DataReader` (abstract)
- `FooDataReader`

3 DCPS Classes and Operations

This chapter describes, for each module, its classes and operations in detail. Each module consists of several classes as defined at PIM level in the DDS-DCPS specification. Some of the classes are implemented as a struct in the PSM. Some of the other classes are abstract, which means they contain some abstract operations.

The Listener interfaces are designed as an interface at PIM level. In other words, the application must implement the interface operations. Therefore, all Listener classes are abstract. A user defined class for these operations must be provided by the application which must extend from the specific Listener class. All Listener operations must be implemented in the user defined class. It is up to the application whether an operation is empty or contains some functionality.

Each class contains several operations, which may be abstract (base class). Abstract operations are not implemented in their base class, but in a type specific class or an application defined class (in case of a Listener). Classes that are implemented as a struct do not have any operations. Some operations are inherited, which means they are implemented in their base class.

The abstract operations in a class are listed (including their synopsis), but not implemented in that class. These operations are implemented in their respective derived classes. The interfaces are fully described, since they must be implemented by the application.

3.1 Infrastructure Module

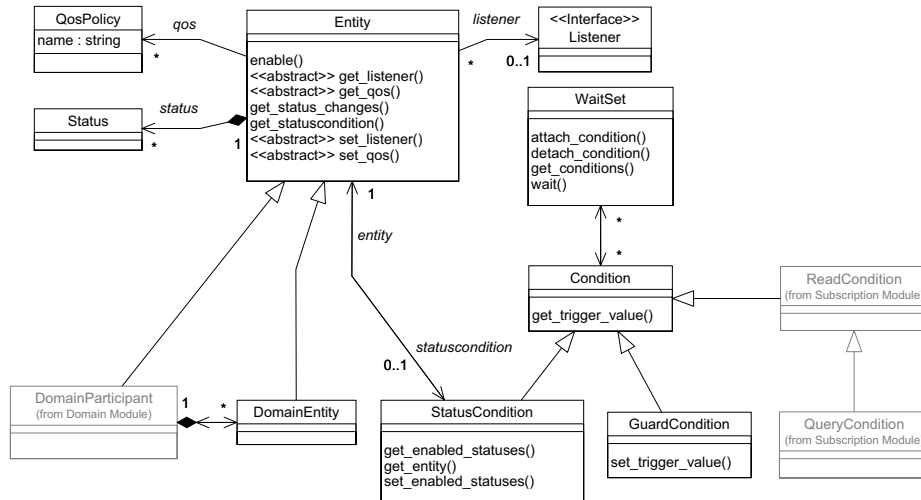


Figure 9 DCPS Infrastructure Module's Class Model

This module contains the following classes:

- Entity (abstract)
- DomainEntity (abstract)
- QosPolicy (abstract, struct)
- Listener (interface)
- Status (abstract, struct)
- WaitSet
- Condition
- GuardCondition
- StatusCondition

3.1.1 Class Entity (abstract)

This class is the abstract base class for all the DCPS objects. It acts as a generic class for Entity objects.

The interface description of this class is as follows:

```
class Entity
{
//
// abstract operations (implemented in class
// DomainParticipant, Topic,
// Publisher, DataWriter, Subscriber and DataReader)
```



```

//
// ReturnCode_t
//   set_qos
//       (const EntityQos& qos);
// ReturnCode_t
//   get_qos
//       (EntityQos& qos);
// ReturnCode_t
//   set_listener
//       (EntityListener_ptr a_listener,
//        StatusMask mask);
// EntityListener_ptr
//   get_listener
//       (void);
//
// implemented API operations
//
    ReturnCode_t
        enable
            (void);
    StatusCondition_ptr
        get_statuscondition
            (void);
    StatusMask
        get_status_changes
            (void);
    InstanceHandle_t
        get_instance_handle
            (void);

};

```

The next paragraphs list all `Entity` operations. The abstract operations are listed but not fully described because they are not implemented in this specific class. The full description of these operations is given in the subclasses, which contain the type specific implementation of these operations.

3.1.1.1 enable

Scope

DDS::Entity

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    enable
        (void);

```

Description

This operation enables the Entity on which it is being called when the Entity was created with the `EntityFactoryQosPolicy` set to `FALSE`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation enables the Entity. Created Entity objects can start in either an enabled or disabled state. This is controlled by the value of the `EntityFactoryQosPolicy` on the corresponding factory for the 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 there 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 Entity does not yet know which QoS settings to apply, thus allowing another piece of code to set the QoS later on. This is for example the case in the DLRL, where the `ObjectHomes` create all underlying DCPS entities but do not know which QoS settings to apply. The user can then apply the required QoS settings afterwards.

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

The `enable` operation is idempotent. Calling `enable` on an already enabled Entity returns `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 `StatusCondition`, the `get_status_changes` operation (although the status of a disabled entity never changes), and the ‘factory’ operations that create, delete or lookup¹ other Entities. Other operations will return the error `RETCODE_NOT_ENABLED`.

Entities created from a factory that is disabled, are created disabled regardless of the setting of the `EntityFactoryQosPolicy`.

1. This includes the `lookup_topicdescription`, but not the `find_topic`.

Calling `enable` on an `Entity` whose factory is not enabled will fail and return `RETCODE_PRECONDITION_NOT_MET`.

If the `EntityFactoryQosPolicy` has `autoenable_created_entities` set to `TRUE`, the `enable` operation on the factory will automatically enable all `Entities` created from the factory.

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 Code

When the operation returns:

- `RETCODE_OK` - the application enabled the `Entity` (or it was already enabled)
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - the factory of the `Entity` is not enabled.

3.1.1.2 `get_instance_handle`

Scope

`DDS::Entity`

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceHandle_t
    get_instance_handle
        (void);
```

Description

This operation returns the `instance_handle` of the builtin topic sample that represents the specified `Entity`.

Parameters

<none>

Return Value

`InstanceHandle_t` - Result value is the `instance_handle` of the builtin topic sample that represents the state of this `Entity`.

Detailed Description

The relevant state of some Entity objects are distributed using builtin topics. Each builtin topic sample represents the state of a specific Entity and has a unique `instance_handle`. This operation returns the `instance_handle` of the builtin topic sample that represents the specified Entity.

Some Entities (Publisher and Subscriber) do not have a corresponding builtin topic sample, but they still have an `instance_handle` that uniquely identifies the Entity. The `instance_handles` obtained this way can also be used to check whether a specific Entity is located in a specific DomainParticipant. (See Section 3.2.1.2, *contains_entity*, on page 112.)

3.1.1.3 get_listener (abstract)

This abstract operation is defined as a generic operation to access a Listener. Each subclass derived from this class, DomainParticipant, Topic, Publisher, Subscriber, DataWriter and DataReader will provide a class specific implementation of this abstract operation.

Synopsis

```
#include <ccpp_dds_dcps.h>
EntityListener_ptr
    get_listener
        (void);
```

3.1.1.4 get_qos (abstract)

This abstract operation is defined as a generic operation to access a struct with the QosPolicy settings. Each subclass derived from this class, DomainParticipant, Topic, Publisher, Subscriber, DataWriter and DataReader will provide a class specific implementation of this abstract operation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (EntityQos& qos);
```

3.1.1.5 get_status_changes

Scope

```
DDS::Entity
```

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
```

```
get_status_changes  
(void);
```

Description

This operation returns a mask with the communication statuses in the `Entity` that are “triggered”.

Parameters

<none>

Return Value

StatusMask - a bit mask in which each bit shows which value has changed.

Detailed Description

This operation returns a mask with the communication statuses in the `Entity` that are *triggered*. That is the set of communication statuses whose value have changed since the last time the application called this operation. This operation shows whether a change has occurred even when the status seems unchanged because the status changed back to the original status.

When the `Entity` is first created or if the `Entity` is not enabled, all communication statuses are in the “un-triggered” state so the mask returned by the operation is empty.

The result value is a bit mask in which each bit shows which value has changed. The relevant bits represent one of the following statuses:

- `INCONSISTENT_TOPIC_STATUS`
- `OFFERED_DEADLINE_MISSED_STATUS`
- `REQUESTED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_ON_READERS_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_LOST_STATUS`
- `LIVELINESS_CHANGED_STATUS`
- `PUBLICATION_MATCHED_STATUS`
- `SUBSCRIPTION_MATCHED_STATUS`

Each status bit is declared as a constant and can be used in an AND operation to check the status bit against the result of type `StatusMask`. Not all statuses are relevant to all `Entity` objects. See the respective `Listener` interfaces for each `Entity` for more information.

3.1.1.6 `get_statuscondition`

Scope

`DDS::Entity`

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

Description

This operation allows access to the `StatusCondition` associated with the `Entity`.

Parameters

<none>

Return Value

StatusCondition - the `StatusCondition` of the `Entity`.

Detailed Description

Each `Entity` has a `StatusCondition` associated with it. This operation allows access to the `StatusCondition` associated with the `Entity`. The returned condition can then be added to a `WaitSet` so that the application can wait for specific status changes that affect the `Entity`.

3.1.1.7 `set_listener` (abstract)

This abstract operation is defined as a generic operation to access a `Listener`. Each subclass derived from this class, `DomainParticipant`, `Topic`, `Publisher`, `Subscriber`, `DataWriter` and `DataReader` will provide a class specific implementation of this abstract operation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_listener
        (EntityListener_ptr a_listener,
```

```
StatusMask mask);
```

3.1.1.8 set_qos (abstract)

This abstract operation is defined as a generic operation to modify the QosPolicy settings. Each subclass derived from this class, DomainParticipant, Topic, Publisher, Subscriber, DataWriter and DataReader will provide a class-specific implementation of this abstract operation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const EntityQos& qos);
```

3.1.2 Class DomainEntity (abstract)

This class is the abstract base class for the all entities except DomainParticipant. The main purpose is to express that DomainParticipant is a special kind of Entity, which acts as a container of all other Entity objects, but cannot contain another DomainParticipant within itself. Therefore, this class is not part of the IDL interface in the DCPS PSM description.

The class DomainEntity does not contain any operations.

3.1.3 Struct QosPolicy

Each Entity provides an <Entity>Qos structure that implements the basic mechanism for an application to specify Quality of Service attributes. This structure consists of Entity specific QosPolicy attributes. QosPolicy attributes are structured types where each type specifies the information that controls an Entity related (configurable) property of the Data Distribution Service.

All QosPolicies applicable to an Entity are aggregated in a corresponding <Entity>Qos, which is a compound structure that is set atomically so that it represents a coherent set of QosPolicy attributes.

Compound types are used whenever multiple attributes must be set coherently to define a consistent attribute for a QosPolicy.

A full description of any <Entity>Qos is given in Appendix A, *Quality Of Service*. The complete list of individual QosPolicy settings and their meaning is described in this paragraph.

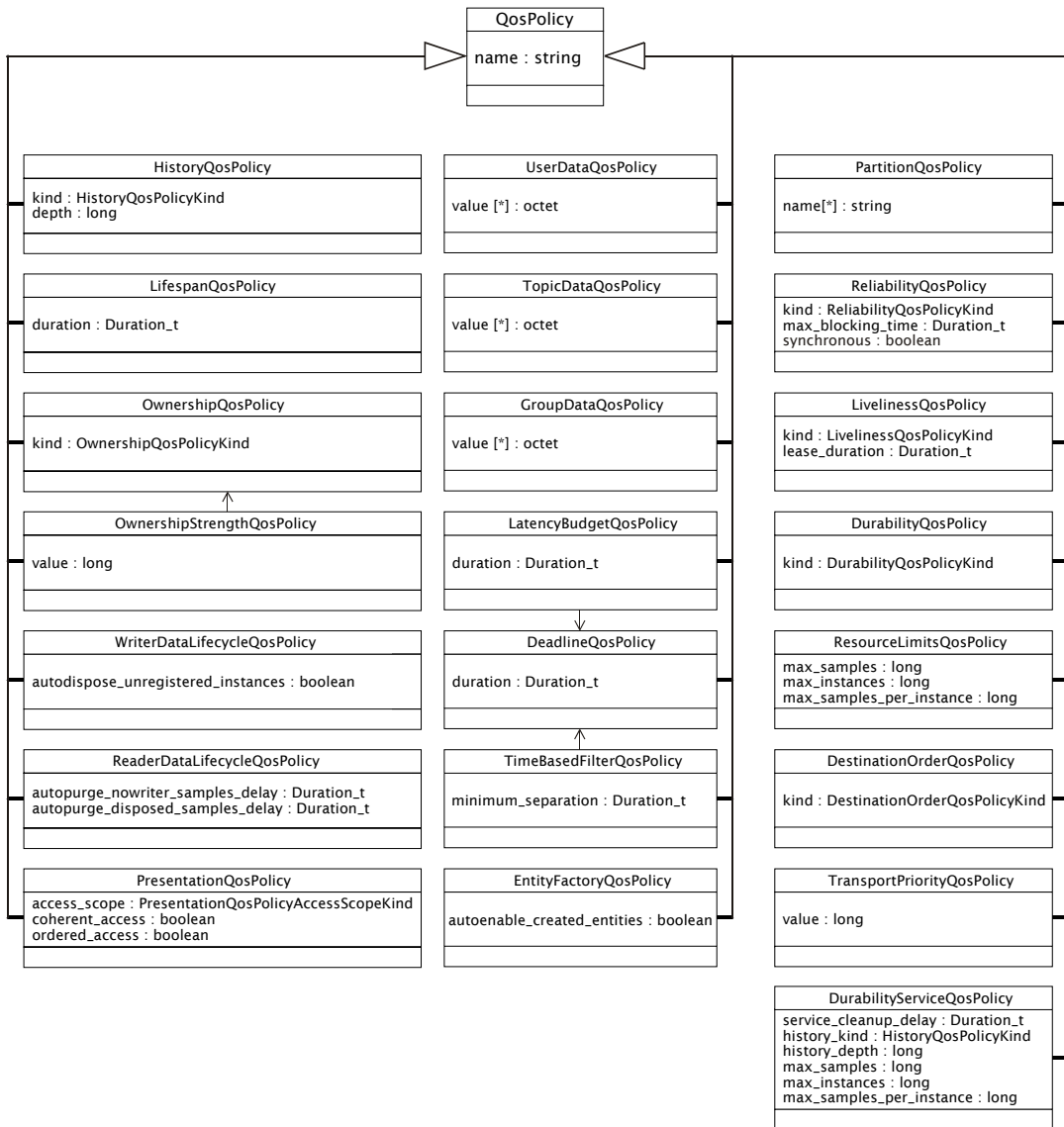


Figure 10 QosPolicy Settings

Requested/Offered

In several cases, for communications to occur properly (or efficiently), a `QosPolicy` on the requesting side must be compatible with a corresponding `QosPolicy` on the offering side. For example, if a `DataReader` requests to receive data reliably while the corresponding `DataWriter` defines a best-effort `QosPolicy`, communication will not happen as requested. This means, the specification for `QosPolicy` follows the Requested/Offered (RxO) pattern while trying to maintain the desirable decoupling of publication and subscription as much as possible. When using this pattern:

- the requesting side can specify a *requested* attribute for a particular `QosPolicy`
- the offering side specifies an *offered* attribute for that `QosPolicy`

The Data Distribution Service will then determine whether the attribute requested by the requesting side is compatible with what is offered by the offering side. Only when the two `QosPolicy` settings are compatible, communication is established. If the two `QosPolicy` settings are not compatible, the Data Distribution Service will not establish communication between the two `Entity` objects and notify this fact by means of the `OFFERED_INCOMPATIBLE_QOS` status on the offering side and the `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side. The application can detect this fact by means of a `Listener` or `Condition`.

The interface description of these `QosPolicies` is as follows:

```
// struct <Entity>Qos
//     see appendix
//
// struct <name>QosPolicy
//
// struct UserDataQosPolicy
//     { OctetSeq value; };
// struct TopicDataQosPolicy
//     { OctetSeq value; };
// struct GroupDataQosPolicy
//     { OctetSeq value; };
// struct TransportPriorityQosPolicy
//     { Long value; };
// struct LifespanQosPolicy
//     { Duration_t duration; };
// enum DurabilityQosPolicyKind
//     { VOLATILE_DURABILITY_QOS,
//       TRANSIENT_LOCAL_DURABILITY_QOS,
//       TRANSIENT_DURABILITY_QOS,
//       PERSISTENT_DURABILITY_QOS };
// struct DurabilityQosPolicy
//     { DurabilityQosPolicyKind kind; };
// enum PresentationQosPolicyAccessScopeKind
//     { INSTANCE_PRESENTATION_QOS,
```

```

        TOPIC_PRESENTATION_QOS,
        GROUP_PRESENTATION_QOS };
struct PresentationQosPolicy
{ PresentationQosPolicyAccessScopeKind access_scope;
  Boolean coherent_access;
  Boolean ordered_access; };
struct DeadlineQosPolicy
{ Duration_t period; };
struct LatencyBudgetQosPolicy
{ Duration_t duration; };
enum OwnershipQosPolicyKind
{ SHARED_OWNERSHIP_QOS,
  EXCLUSIVE_OWNERSHIP_QOS };
struct OwnershipQosPolicy
{ OwnershipQosPolicyKind kind; };
struct OwnershipStrengthQosPolicy
{ Long value; };
enum LivelinessQosPolicyKind
{ AUTOMATIC_LIVELINESS_QOS,
  MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
  MANUAL_BY_TOPIC_LIVELINESS_QOS };
struct LivelinessQosPolicy
{ LivelinessQosPolicyKind kind;
  Duration_t lease_duration; };
struct TimeBasedFilterQosPolicy
{ Duration_t minimum_separation; };
struct PartitionQosPolicy
{ StringSeq name; };
enum ReliabilityQosPolicyKind
{ BEST_EFFORT_RELIABILITY_QOS,
  RELIABLE_RELIABILITY_QOS };
struct ReliabilityQosPolicy
{ ReliabilityQosPolicyKind kind;
  Duration_t max_blocking_time;
  Boolean synchronous; };
enum DestinationOrderQosPolicyKind
{ BY_RECEPTION_timestamp_DESTINATIONORDER_QOS,
  BY_SOURCE_timestamp_DESTINATIONORDER_QOS };
struct DestinationOrderQosPolicy
{ DestinationOrderQosPolicyKind kind; };
enum HistoryQosPolicyKind
{ KEEP_LAST_HISTORY_QOS,
  KEEP_ALL_HISTORY_QOS };
struct HistoryQosPolicy
{ HistoryQosPolicyKind kind;
  Long depth; };
struct ResourceLimitsQosPolicy
{ Long max_samples;
  Long max_instances;
  Long max_samples_per_instance; };

```

```

struct EntityFactoryQosPolicy
{ Boolean autoenable_created_entities; };
struct WriterDataLifecycleQosPolicy
{ Boolean autodispose_unregistered_instances; };
struct ReaderDataLifecycleQosPolicy
{ Duration_t autopurge_nowriter_samples_delay;
  Duration_t autopurge_disposed_samples_delay; };
struct DurabilityServiceQosPolicy
{ Duration_t service_cleanup_delay;
  HistoryQosPolicyKind history_kind;
  Long history_depth;
  Long max_samples;
  Long max_instances;
  Long max_samples_per_instance; };
enum SchedulingClassQosPolicyKind
{ SCHEDULE_DEFAULT,
  SCHEDULE_TIMESHARING,
  SCHEDULE_REALTIME };
struct SchedulingClassQosPolicy
{ SchedulingClassQosPolicyKind kind; };
enum SchedulingPriorityQosPolicyKind
{ PRIORITY_RELATIVE,
  PRIORITY_ABSOLUTE };
struct SchedulingPriorityQosPolicy
{ SchedulingPriorityQosPolicyKind kind; };
struct SchedulingQosPolicy
{ SchedulingClassQosPolicy scheduling_class;
  SchedulingPriorityQosPolicy scheduling_priority_kind;
  long scheduling_priority; };

```

Default Attributes

The default attributes of each QosPolicy are listed in the next table:

Table 2 QosPolicy Default Attributes

QosPolicy	Attribute	Value
user_data	value.length	0
topic_data	value.length	0
group_data	value.length	0
transport_priority	value	0
lifespan	duration	DURATION_INFINITE
durability	kind	VOLATILE_DURABILITY_QOS
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	FALSE
	ordered_access	FALSE

Table 2 QosPolicy Default Attributes (Continued)

QosPolicy	Attribute	Value
deadline	period	DURATION_INFINITE
latency_budget	duration	0
ownership_strength	value	0
ownership	kind	SHARED_OWNERSHIP_QOS
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITE
time_based_filter	minimum_separation	0
partition	name.length	0
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	FALSE
destination_order	kind	BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
entity_factory	autoenable_created_entities	TRUE
writer_data_lifecycle	autodispose_unregistered_instances	TRUE
reader_data_lifecycle	autopurge_nowriter_samples_delay	DURATION_INFINITE
	autopurge_disposed_samples_delay	DURATION_INFINITE
durability_service	history_kind	KEEP_LAST
	history_depth	1
	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
	service_cleanup_delay	0

Table 2 QosPolicy Default Attributes (Continued)

QosPolicy	Attribute	Value
watchdog_scheduling, listener_scheduling	scheduling_class.kind	SCHEDULE_DEFAULT
	scheduling_priority_kind. kind	PRIORITY_RELATIVE
	scheduling_priority	0

RxO

The `QosPolicy` settings that need to be set in a compatible manner between the publisher and subscriber ends are indicated by the setting of the “RxO” (Requested/Offered) property. The `RxO` property of each `QosPolicy` is listed in Table 3 on page 40. Please note:

- A `RxO` setting of `Yes` indicates that the `QosPolicy` can be set at both ends (publishing and subscribing) and the attributes must be set in a compatible manner. In this case the compatible attributes are explicitly defined.
- A `RxO` setting of `No` indicates that the `QosPolicy` can be set at both ends (publishing and subscribing) but the two settings are independent. That is, all combinations of attributes are compatible.
- A `RxO` setting of `Not applicable` indicates that the `QosPolicy` can only be specified at either the publishing or the subscribing end, but not at both ends. So compatibility does not apply.

Changeable

The `changeable` property determines whether the `QosPolicy` can be changed after the `Entity` is enabled. In other words, a `QosPolicy` with `changeable` setting of `No` is considered “immutable” and can only be specified either at `Entity` creation time or prior to calling the enable operation on the `Entity`.

When the application tries to change a `QosPolicy` with `changeable` setting of `No`, the Data Distribution Service will notify this by returning a `RETCODE_IMMUTABLE_POLICY`.

The basic way to modify or set the `<Entity>Qos` is by using a `get_qos` and `set_qos` operation to get all `QosPolicy` settings from this `Entity` (that is the `<Entity>Qos`), modify several specific `QosPolicy` settings and put them back using an user operation to set all `QosPolicy` settings on this `Entity` (that is the `<Entity>Qos`). An example of these operations for the `DataWriter` are `get_qos` and `set_qos`, which take the `<Entity>Qos` as a parameter.

The “RxO” setting and the “changeable” setting of each QosPolicy are listed in the next table:

Table 3 QosPolicy Basics

QosPolicy	Concerns Entity	RxO	Changeable After Enabling
user_data	DomainParticipant DataReader DataWriter	No	Yes
topic_data	Topic	No	Yes
group_data	Publisher Subscriber	No	Yes
transport_priority	Topic DataWriter	Not applicable	Yes
lifespan	Topic DataWriter	Not applicable	Yes
durability	Topic DataReader DataWriter	Yes	No
presentation	Publisher Subscriber	Yes	No
deadline	Topic DataReader DataWriter	Yes	Yes
latency_budget	Topic DataReader DataWriter	Yes	Yes
ownership	Topic DataReader DataWriter	Yes	No
ownership_strength	DataWriter	Not applicable	Yes
liveliness	Topic DataReader DataWriter	Yes	No
time_based_filter	DataReader	Not applicable	Yes

Table 3 QosPolicy Basics (Continued)

QosPolicy	Concerns Entity	RxO	Changeable After Enabling
partition	Publisher Subscriber	No	Yes
reliability	Topic DataReader DataWriter	Yes	No
destination_order	Topic DataReader DataWriter	Yes	No
history	Topic DataReader DataWriter	No	No
resource_limits	Topic DataReader DataWriter	No	No
entity_factory	DomainParticipantFactory DomainParticipant Publisher Subscriber	No	Yes
writer_data_lifecycle	DataWriter	Not applicable	Yes
reader_data_lifecycle	DataReader	Not applicable	Yes
durability_service	Topic	No	No
scheduling	DomainParticipant	Not applicable	No

The next paragraphs describe the usage of each <name>QosPolicy struct.

3.1.3.1 DeadlineQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct DeadlineQosPolicy
{ Duration_t period; };
```

Description

This `QosPolicy` defines the period within which a new sample is expected by the `DataReader` or to be written by the `DataWriter`.

Attributes

Duration_t period - specifies the period within which a new sample is expected or to be written.

Detailed Description

This `QosPolicy` will set the period within which a `DataReader` expects a new sample or, in case of a `DataWriter`, the period in which it expects applications to write the sample. The default value of the period is `DURATION_INFINITE`, indicating that there is no deadline. The `QosPolicy` may be used to monitor the real-time behaviour, a `Listener` or a `StatusCondition` may be used to catch the event that is generated when a deadline is missed.

`DeadlineQosPolicy` is instance oriented (i.e. the period is monitored for each individual instance).

The exact consequences of a missed deadline depend on the Entity in which it occurred, and the `OwnershipQosPolicy` value of that Entity:

- In case a `DataWriter` misses an instance deadline (regardless of its `OwnershipQosPolicy` setting), an `offered_deadline_missed_status` is raised, which can be detected by either a `Listener` or a `StatusCondition`. There are no further consequences.
- In case a `DataReader` misses an instance deadline, a `requested_deadline_missed_status` is raised, which can be detected by either a `Listener` or a `StatusCondition`. In case the `OwnershipQosPolicy` is set to `SHARED`, there are no further consequences. In case the `OwnershipQosPolicy` is set to `EXCLUSIVE`, the ownership of that instance on that particular `DataReader` is transferred to the next available highest strength `DataWriter`, but this will have no impact on the `instance_state` whatsoever. So even when a deadline is missed for an instance that has no other (lower-strength) `DataWriters` to transfer ownership to, the `instance_state` remains unchanged. See also Section 3.1.3.11, *OwnershipQosPolicy*.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning Entity, this `QosPolicy` may be changed by using the `set_qos` operation.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 4 DeadlineQosPolicy

Period	Compatibility
offered period < requested period	compatible
offered period = requested period	compatible
offered period > requested period	Incompatible

Whether communication is established, is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. In other words, the communication between any `DataWriter` and `DataReader` depends on what is expected by the `DataReader`. As a consequence, a `DataWriter` that has an incompatible `QoS` with respect to what a `DataReader` specifies, is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible `QoS` with respect to what a `DataWriter` specifies, does not get any data from that particular `DataWriter`.

Changing an existing deadline period using the `set_qos` operation on either the `DataWriter` or `DataReader` may have consequences for the connectivity between readers and writers, depending on their `RxO` values. (See also in Section 3.1.3, *Struct QosPolicy*, the paragraph entitled *Requested/Offered*.) Consider a writer with deadline period P_w and a reader with deadline period P_r , where $P_w \leq P_r$. In this case a connection between that reader and that writer is established. Now suppose P_w is changed so that $P_w > P_r$, then the existing connection between reader and writer will be lost, and the reader will behave as if the writer unregistered all its instances, transferring the ownership of these instances when appropriate. See also Section 3.1.3.11, *OwnershipQosPolicy*.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this `qos` by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.2 DestinationOrderQosPolicy**Scope**

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum DestinationOrderQosPolicyKind
{ BY_RECEPTION_timestamp_DESTINATIONORDER_QOS,
  BY_SOURCE_timestamp_DESTINATIONORDER_QOS };
struct DestinationOrderQosPolicy
{ DestinationOrderQosPolicyKind kind; };
```

Description

This `QosPolicy` controls the order in which the `DataReader` stores the data.

Attributes

DestinationOrderQosPolicyKind kind - controls the order in which the `DataReader` stores the data.

Detailed Description

This `QosPolicy` controls the order in which the `DataReader` stores the data. The order of storage is controlled by the timestamp. However a choice can be made to use the timestamp of the `DataReader` (time of reception) or the timestamp of the `DataWriter` (source timestamp).

This `QosPolicy` is applicable to a `DataWriter`, `DataReader` and a `Topic`. After enabling of the concerning entity, this `QosPolicy` cannot be changed any more.

Attribute

The `QosPolicy` is controlled by the attribute `kind` which may be:

- `BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS`
- `BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS`

When set to `BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS`, the order is based on the timestamp, at the moment the sample was received by the `DataReader`.

When set to `BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS`, the order is based on the timestamp, which was set by the `DataWriter`. This means that the system needs some time synchronisation.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 5 Requested/Offered DestinationOrderQosPolicy

<div>Requested Offered</div>	BY_RECEPTION _timestamp	BY_SOURCE_tim estamp
BY_RECEPTION_timestamp	compatible	Incompatible
BY_SOURCE_timestamp	compatible	compatible

Whether communication is established, is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. In other words, the communication between any `DataWriter` and `DataReader` depends on what is expected by the `DataReader`. As a consequence, a `DataWriter` that has an incompatible `QoS` with respect to what a `DataReader` specified, is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible `QoS` with respect to what a `DataWriter` specified, does not get any data from that particular `DataWriter`.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this `qos` by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.3 DurabilityQosPolicy**Scope**

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum DurabilityQosPolicyKind
{ VOLATILE_DURABILITY_QOS,
  TRANSIENT_LOCAL_DURABILITY_QOS,
  TRANSIENT_DURABILITY_QOS,
  PERSISTENT_DURABILITY_QOS };
struct DurabilityQosPolicy
{ DurabilityQosPolicyKind kind; };
```

Description

This `QosPolicy` controls whether the data should be stored for late joining readers.

Attributes

DurabilityQosPolicyKind kind - specifies the type of durability from `VOLATILE_DURABILITY_QOS` (short life) to `PERSISTENT_DURABILITY_QOS` (long life).

Detailed Description

The decoupling between `DataReader` and `DataWriter` offered by the Data Distribution Service allows an application to write data even if there are no current readers on the network. Moreover, a `DataReader` that joins the network after some data has been written could potentially be interested in accessing the most current values of the data as well as some history. This `QosPolicy` controls whether the Data Distribution Service will actually make data available to late-joining `DataReaders`.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` cannot be changed any more.

Attributes

The `QosPolicy` is controlled by the attribute `kind` which may be:

- `VOLATILE_DURABILITY_QOS` - the samples are not available to late-joining `DataReaders`. In other words, only `DataReaders`, which were present at the time of the writing and have subscribed to this `Topic`, will receive the sample. When a `DataReader` subscribes afterwards (late-joining), it will only be able to read the next written sample. This setting is typically used for data, which is updated quickly
- `TRANSIENT_LOCAL_DURABILITY_QOS` - this is currently implemented as for the `TRANSIENT_DURABILITY_QOS`. The desired behaviour of `TRANSIENT_LOCAL_DURABILITY_QOS` can be achieved from the `TRANSIENT_DURABILITY_QOS` with the default (`TRUE`) setting of the `autodispose_unregistered_instances` flag on the `DataWriter`. This is because for `TRANSIENT_LOCAL`, the data is required to be kept in the memory of the `DataWriter` that wrote the data, but not required to survive it.
- `TRANSIENT_DURABILITY_QOS` - some samples are available to late-joining `DataReaders` (stored in memory). This means that the late-joining `DataReaders` are able to read these previously written samples. The `DataReader` does not necessarily have to exist at the time of writing. Not all samples are stored (depending on `QosPolicy History` and `QosPolicy resource_limits`). The storage does not depend on the `DataWriter` and will outlive the `DataWriter`. This may be used to implement reallocation of applications because the data is saved in the Data Distribution Service (not in the

DataWriter). This setting is typically used for state related information of an application. In this case also the `DurabilityServiceQosPolicy` settings are relevant for the behaviour of the Data Distribution Service

- `PERSISTENT_DURABILITY_QOS` - the data is stored in permanent storage (e.g. hard disk). This means that the samples are also available after a system restart. The samples not only outlives the `DataWriters`, but even the Data Distribution Service and the system. This setting is typically used for attributes and settings for an application or the system. In this case also the `DurabilityServiceQosPolicy` settings are relevant for the behaviour of the Data Distribution Service.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 6 Requested/Offered DurabilityQosPolicy

<div>Requested Offered</div>	VOLATILE	TRANSIENT	PERSISTENT
VOLATILE	compatible	incompatible	INcompatible
TRANSIENT	compatible	compatible	INcompatible
PERSISTENT	compatible	compatible	compatible

This means that the Request/Offering mechanism is applicable between:

- the `DataWriter` and the `DataReader`. If the `QosPolicy` settings between `DataWriter` and `DataReader` are inconsistent, no communication between them is established. In addition the `DataWriter` will be informed via a `REQUESTED_INCOMPATIBLE_QOS` status change and the `DataReader` will be informed via an `OFFERED_INCOMPATIBLE_QOS` status change
- the `DataWriter` and the Data Distribution Service (as a built-in `DataReader`). If the `QosPolicy` settings between `DataWriter` and the Data Distribution Service are inconsistent, no communication between them is established. In that case data published by the `DataWriter` will not be maintained by the service and as a consequence will not be available for late joining `DataReaders`. The `QosPolicy` of the Data Distribution Service in the role of `DataReader` is specified by the `Topic QosPolicy`
- the Data Distribution Service (as a built-in `DataWriter`) and the `DataReader`. If the `QosPolicy` settings between the Data Distribution Service and the `DataReader` are inconsistent, no communication between them is established. In

that case the Data Distribution Service will not publish historical data to late joining DataReaders. The QosPolicy of the Data Distribution Service in the role of DataWriter is specified by the Topic QosPolicy.

Cleanup

The DurabilityQosPolicy kind setting TRANSIENT_DURABILITY_QOS and PERSISTENT_DURABILITY_QOS determine that the DurabilityServiceQosPolicy applies for the Topic. It controls amongst others at which time the durability service is allowed to remove all information regarding a data-instance. Information on a data-instance is maintained until the following conditions are met:

- the instance has been explicitly disposed of (`instance_state = NOT_ALIVE_DISPOSED_INSTANCE_STATE`),
- *and* the system detects that there are no more “live” DataWriter objects writing the instance, that is, all DataWriter either `unregister_instance` the instance (call `unregister_instance` operation) or lose their liveliness,
- *and* a time interval longer than `service_cleanup_delay` has elapsed since the moment the Data Distribution Service detected that the previous two conditions were met.

The use of the DurabilityServiceQosPolicy attribute `service_cleanup_delay` is apparent in the situation where an application disposes of an instance and it crashes before having a chance to complete additional tasks related to the disposition. Upon re-start the application may ask for initial data to regain its state and the delay introduced by the `service_cleanup_delay` allows the re-started application to receive the information on the disposed of instance and complete the interrupted tasks.

TopicQos

This QosPolicy can be set on a Topic. The DataWriter and/or DataReader can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the QosPolicy for the Topic, DataReader and DataWriter are consistent.

3.1.3.4 DurabilityServiceQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct DurabilityServiceQosPolicy
```

```
{ Duration_t service_cleanup_delay;  
  HistoryQosPolicyKind history_kind;  
  Long history_depth;  
  Long max_samples;  
  Long max_instances;  
  Long max_samples_per_instance; };
```

Description

This `QosPolicy` controls the behaviour of the durability service regarding transient and persistent data.

Attributes

Duration_t service_cleanup_delay - specifies how long the durability service must wait before it is allowed to remove the information on the transient or persistent topic data-instances as a result of incoming dispose messages.

HistoryQosPolicyKind history_kind - specifies the type of history, which may be `KEEP_LAST_HISTORY_QOS` or `KEEP_ALL_HISTORY_QOS`, the durability service must apply for the transient or persistent topic data-instances.

Long history_depth - specifies the number of samples of each instance of data (identified by its key) that is managed by the durability service for the transient or persistent topic data-instances. If *history_kind* is `KEEP_LAST_HISTORY_QOS`, *history_depth* must be smaller than or equal to *max_samples_per_instance* for this `QosPolicy` to be consistent.

Long max_samples - specifies the maximum number of data samples for all instances the durability service will manage for the transient or persistent topic data-instances.

Long max_instances - specifies the maximum number of instances the durability service - manage for the transient or persistent topic data-instances.

Long max_samples_per_instance - specifies the maximum number of samples of any single instance the durability service will manage for the transient or persistent topic data-instances. If *history_kind* is `KEEP_LAST_HISTORY_QOS`, *max_samples_per_instance* must be greater than or equal to *history_depth* for this `QosPolicy` to be consistent.

Detailed Description

This `QosPolicy` controls the behaviour of the durability service regarding transient and persistent data. It controls for the transient or persistent topic; the time at which information regarding the topic may be discarded, the history policy it must set and the resource limits it must apply.

Cleanup

The setting of the `DurabilityServiceQosPolicy` only applies when kind of the `DurabilityQosPolicy` is either `TRANSIENT_DURABILITY_QOS` or `PERSISTENT_DURABILITY_QOS`. The `service_cleanup_delay` setting controls at which time the durability service” is allowed to remove all information regarding a data-instance. Information on a data-instance is maintained until the following conditions are met:

- the instance has been explicitly disposed of (`instance_state = NOT_ALIVE_DISPOSED_INSTANCE_STATE`),
- *and* the system detects that there are no more “live” `DataWriter` objects writing the instance, that is, all `DataWriter` either `unregister_instance` the instance (call `unregister_instance` operation) or lose their liveliness,
- *and* a time interval longer than `service_cleanup_delay` has elapsed since the moment the Data Distribution Service detected that the previous two conditions were met.

The use of the attribute `service_cleanup_delay` is apparent in the situation where an application disposes of an instance and it crashes before having a chance to complete additional tasks related to the disposition. Upon re-start the application may ask for initial data to regain its state and the delay introduced by the `service_cleanup_delay` allows the re-started application to receive the information on the disposed of instance and complete the interrupted tasks.

History

The attributes `history_kind` and `history_depth` apply to the history settings of the durability service’s internal `DataWriter` and `DataReader` managing the topic. The `HistoryQosPolicy` behaviour, as described in paragraph 3.1.3.7 (`HistoryQosPolicy`), applies to these attributes.

Resource Limits

The attributes `max_samples`, `max_instances` and `max_samples_per_instance` apply to the resource limits of the Durability Service’s internal `DataWriter` and `DataReader` managing the topic. The `ResourceLimitsQosPolicy` behaviour, as described in paragraph 3.1.3.17 (`ResourceLimitsQosPolicy`) applies to these attributes.

TopicQos

This `QosPolicy` can be set on a `Topic` only. After enabling of the concerning `Topic`, this `QosPolicy` can not be changed any more.

3.1.3.5 EntityFactoryQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct EntityFactoryQosPolicy
{ Boolean autoenable_created_entities; };
```

Description

This QosPolicy controls the behaviour of the Entity as a factory for other entities.

Attributes

Boolean autoenable_created_entities - specifies whether the entity acting as a factory automatically enables the instances it creates. If *autoenable_created_entities* is *TRUE* the factory will automatically enable each created Entity, otherwise it will not.

Detailed Description

This QosPolicy controls the behaviour of the Entity as a factory for other entities. It concerns only *DomainParticipantFactory* (as factory for *DomainParticipant*), *DomainParticipant* (as factory for *Publisher*, *Subscriber*, and *Topic*), *Publisher* (as factory for *DataWriter*), and *Subscriber* (as factory for *DataReader*).

This policy is mutable. A change in the policy affects only the entities created after the change; not the previously created entities.

The setting of *autoenable_created_entities* to *TRUE* indicates that the factory *create_<entity>* operation will automatically invoke the *enable* operation each time a new Entity is created. Therefore, the Entity returned by *create_<entity>* will already be enabled. A setting of *FALSE* indicates that the Entity will not be automatically enabled: the application will need to enable it explicitly by means of the *enable* operation. See Section 3.1.1.1, *enable*, for a detailed description about the differences between enabled and disabled entities.

The default setting of *autoenable_created_entities* is *TRUE* meaning that by default it is not necessary to explicitly call *enable* on newly created entities.

3.1.3.6 GroupDataQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct GroupDataQosPolicy
{ OctetSeq value; };
```

Description

This `QosPolicy` allows the application to attach additional information to a Publisher or Subscriber Entity. This information is distributed with the `BuiltinTopics`.

Attributes

OctetSeq value - a sequence of octets that holds the application group data. By default, the sequence has length 0.

Detailed Description

This `QosPolicy` allows the application to attach additional information to a Publisher or Subscriber Entity. This information is distributed with the `BuiltinTopic`. An application that discovers a new Entity of the listed kind, can use this information to add additional functionality. The `GroupDataQosPolicy` is changeable and updates of the `BuiltinTopic` instance must be expected. Note that the Data Distribution Service is not aware of the real structure of the group data (the Data Distribution System handles it as an opaque type) and that the application is responsible for correct mapping on structural types for the specific platform.

3.1.3.7 HistoryQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum HistoryQosPolicyKind
{ KEEP_LAST_HISTORY_QOS,
  KEEP_ALL_HISTORY_QOS };
struct HistoryQosPolicy
{ HistoryQosPolicyKind kind;
  Long depth; };
```

Description

This `QosPolicy` controls which samples will be stored when the value of an instance changes (one or more times) before it is finally communicated.

Attributes

HistoryQosPolicyKind kind - specifies the type of history, which may be `KEEP_LAST_HISTORY_QOS` or `KEEP_ALL_HISTORY_QOS`.

Long depth - specifies the number of samples of each instance of data (identified by its key) managed by this Entity.

Detailed Description

This `QosPolicy` controls whether the Data Distribution Service should deliver only the most recent sample, attempt to deliver all samples, or do something in between. In other words, how the `DataWriter` or `DataReader` should store samples. Normally, only the most recent sample is available but some history can be stored.

DataWriter

On the publishing side this `QosPolicy` controls the samples that should be maintained by the `DataWriter` on behalf of existing `DataReader` objects. The behaviour with respect to a `DataReader` objects discovered after a sample is written is controlled by the `DurabilityQosPolicy`.

DataReader

On the subscribing side it controls the samples that should be maintained until the application “takes” them from the Data Distribution Service.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning Entity, this `QosPolicy` cannot be changed any more.

Attributes

The `QosPolicy` is controlled by the attribute `kind` which can be:

- `KEEP_LAST_HISTORY_QOS` - the Data Distribution Service will only attempt to keep the latest values of the instance and discard the older ones. The attribute “depth” determines how many samples in history will be stored. In other words, only the most recent samples in history are stored. On the publishing side, the Data Distribution Service will only keep the most recent “depth” samples of each instance of data (identified by its key) managed by the `DataWriter`. On the subscribing side, the `DataReader` will only keep the most recent “depth” samples received for each instance (identified by its key) until the application “takes” them via the `DataReader::take` operation. `KEEP_LAST_HISTORY_QOS` - is the default `kind`. The default value of `depth` is 1, indicating that only the most recent value should be delivered. If a `depth` other than 1 is specified, it should be compatible with the settings of the `ResourceLimitsQosPolicy max_samples_per_instance`. For these two

`QosPolicy` settings to be compatible, they must verify that `depth <= max_samples_per_instance`, otherwise a `RETCODE_INCONSISTENT_POLICY` is generated on relevant operations

- `KEEP_ALL_HISTORY_QOS` - all samples are stored, provided, the resources are available. On the publishing side, the Data Distribution Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its key) managed by the `DataWriter` until they can be delivered to all subscribers. On the subscribing side, the Data Distribution Service will attempt to keep all samples of each instance of data (identified by its key) managed by the `DataReader`. These samples are kept until the application “takes” them from the Data Distribution Service via the `DataReader::take` operation. The setting of `depth` has no effect. Its implied value is `LENGTH_UNLIMITED`. The resources that the Data Distribution Service can use to keep this history are limited by the settings of the `ResourceLimitsQosPolicy`. If the limit is reached, the behaviour of the Data Distribution Service will depend on the `ReliabilityQosPolicy`. If the `ReliabilityQosPolicy` is `BEST_EFFORT_RELIABILITY_QOS`, the old values are discarded. If `ReliabilityQosPolicy` is `RELIABLE_RELIABILITY_QOS`, the Data Distribution Service will block the `DataWriter` until it can deliver the necessary old values to all subscribers.

On the subscribing side it controls the samples that should be maintained until the application “takes” them from the Data Distribution Service. On the publishing side this `QosPolicy` controls the samples that should be maintained by the `DataWriter` on behalf of `DataReader` objects. The behaviour with respect to a `DataReader` objects discovered after a sample is written is controlled by the `DurabilityQosPolicy`. In more detail, this `QosPolicy` specifies the behaviour of the Data Distribution Service in case the value of a sample changes (one or more times) before it can be successfully communicated to one or more Subscribers.

Requested/Offered

The setting of the `QosPolicy` offered is independent of the one requested, in other words they are never considered incompatible. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side or `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side will not be raised.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.8 LatencyBudgetQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct LatencyBudgetQosPolicy
{ Duration_t duration; };
```

Description

Specifies the maximum acceptable additional delay to the typical transport delay from the time the data is written until the data is delivered at the `DataReader` and the application is notified of this fact.

Attributes

Duration_t duration - specifies the maximum acceptable additional delay from the time the data is written until the data is delivered.

Detailed Description

This `QosPolicy` specifies the maximum acceptable additional delay to the typical transport delay from the time the data is written until the data is delivered at the `DataReader` and the application is notified of this fact. This `QosPolicy` provides a means for the application to indicate to the Data Distribution Service the “urgency” of the data-communication. By having a non-zero duration the Data Distribution Service can optimise its internal operation. The default value of the duration is zero, indicating that the delay should be minimized.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning Entity, this `QosPolicy` may be changed by using the `set_qos` operation.

Requested/Offered

This `QosPolicy` is considered a hint to the Data Distribution Service, which will automatically adapt its behaviour to meet the requirements of the shortest delay if possible. In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 7 LatencyBudgetQosPolicy

Duration	Compatibility
offered duration < requested duration	compatible

Table 7 LatencyBudgetQosPolicy(Continued)

Duration	Compatibility
offered duration = requested duration	compatible
offered duration > requested duration	INcompatible

Note that even when the offered duration is considered compatible to the requested duration, this duration is not enforced in any way: there will be no notification on any violations of the requested duration.

Changing an existing latency budget using the `set_qos` operation on either the `DataWriter` or `DataReader` may have consequences for the connectivity between readers and writers, depending on their `RxO` values. (See also in Section 3.1.3, *Struct QosPolicy*, the paragraph entitled *Requested/Offered*.) Consider a writer with budget B_w and a reader with budget B_r , where $B_w \leq B_r$. In this case a connection between that reader and that writer is established. Now suppose B_w is changed so that $B_w > B_r$, then the existing connection between reader and writer will be lost, and the reader will behave as if the writer unregistered all its instances, transferring the ownership of these instances when appropriate. See also Section 3.1.3.11, *OwnershipQosPolicy*.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this `qos` by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.9 LifespanQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
    struct LifespanQosPolicy
    { Duration_t duration; };
```

Description

This `QosPolicy` specifies the duration of the validity of the data written by the `DataWriter`.

Attributes

Duration_t duration - specifies the length in time of the validity of the data.

Detailed Description

This `QosPolicy` specifies the duration of the validity of the data written by the `DataWriter`. When this time has expired, the data will be removed or if it has not been delivered yet, it will not be delivered at all. In other words, the *duration* is the time in which the data is still valid. This means that during this period a `DataReader` can access the data or if the data has not been delivered yet, it still will be delivered. The default value of the *duration* is `DURATION_INFINITE`, indicating that the data does not expire.

This `QosPolicy` is applicable to a `DataWriter` and a `Topic`. After enabling of the concerning `Entity`, this `QosPolicy` may be changed by using the `set_qos` operation.

Requested/Offered

The setting of this `QosPolicy` is only applicable to the publishing side, in other words the Requested/Offered constraints are not applicable. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side will not be raised.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.10 LivelinessQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum LivelinessQosPolicyKind
{
    AUTOMATIC_LIVELINESS_QOS,
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
    MANUAL_BY_TOPIC_LIVELINESS_QOS };
struct LivelinessQosPolicy
{
    LivelinessQosPolicyKind kind;
    Duration_t lease_duration; };

```

Description

This QosPolicy controls the way the liveliness of an Entity is being determined.

Attributes

LivelinessQosPolicyKind kind - controls the way the liveliness of an Entity is determined.

Duration_t lease_duration - specifies the duration of the interval within which the liveliness must be reported.

Detailed Description

This QosPolicy controls the way the liveliness of an Entity is being determined. The liveliness must be reported periodically before the *lease_duration* expires.

This QosPolicy is applicable to a DataReader, a DataWriter and a Topic. After enabling of the concerning Entity, this QosPolicy cannot be changed any more.

Attributes

The QosPolicy is controlled by the attribute *kind* which can be:

- *AUTOMATIC_LIVELINESS_QOS* - the Data Distribution Service will take care of reporting the Liveliness automatically with a rate determined by the *lease_duration*.
- *MANUAL_BY_PARTICIPANT_LIVELINESS_QOS* - the application must take care of reporting the liveliness before the *lease_duration* expires. If an Entity reports its liveliness, all Entities within the same DomainParticipant that have their liveliness kind set to *MANUAL_BY_PARTICIPANT_LIVELINESS_QOS*, can be considered alive by the Data Distribution Service. Liveliness can be reported explicitly by calling the operation *assert_liveliness* on the DomainParticipant or implicitly by writing some data.
- *MANUAL_BY_TOPIC_LIVELINESS_QOS* - the application must take care of reporting the liveliness before the *lease_duration* expires. This can explicitly be done by calling the operation *assert_liveliness* on the DataWriter or implicitly by writing some data.

The *lease_duration* specifies the duration of the interval within which the liveliness should be reported.

Requested/Offered

In case the Requested/Offered QosPolicy are incompatible, the notification *OFFERED_INCOMPATIBLE_QOS* status on the offering side and *REQUESTED_INCOMPATIBLE_QOS* status on the requesting side is raised.

Table 8 LivelinessQosPolicy

<div>Requested Offered</div>	AUTOMATIC	MANUAL_BY_ PARTICIPANT	MANUAL_BY_ TOPIC
AUTOMATIC	compatible	INcompatible	INcompatible
MANUAL_BY_PARTICIPANT	compatible	compatible	incompatible
MANUAL_BY_TOPIC	compatible	compatible	compatible

Whether communication is established, is controlled by the Data Distribution Service, depending on the Requested/Offered QoSPolicy of the DataWriter and DataReader. In other words, the communication between any DataWriter and DataReader depends on what is expected by the DataReader. As a consequence, a DataWriter that has an incompatible QoS with respect to what a DataReader specified is not allowed to send its data to that specific DataReader. A DataReader that has an incompatible QoS with respect to what a DataWriter specified does not get any data from that particular DataWriter.

TopicQos

This QoSPolicy can be set on a Topic. The DataWriter and/or DataReader can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the QoSPolicy for the Topic, DataReader and DataWriter are consistent.

3.1.3.11 OwnershipQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum OwnershipQosPolicyKind
{ SHARED_OWNERSHIP_QOS,
  EXCLUSIVE_OWNERSHIP_QOS };
struct OwnershipQosPolicy
{ OwnershipQosPolicyKind kind; };
```

Description

This QoSPolicy specifies whether a DataWriter exclusively owns an instance.

Attributes

OwnershipQosPolicyKind kind - specifies whether a DataWriter exclusively owns an instance.

Detailed Description

This `QosPolicy` specifies whether a `DataWriter` exclusively may own an instance. In other words, whether multiple `DataWriter` objects can write the same instance at the same time. The `DataReader` objects will only read the modifications on an instance from the `DataWriter` owning the instance.

Exclusive ownership is on an instance-by-instance basis. That is, a `Subscriber` can receive values written by a lower strength `DataWriter` as long as they affect instances whose values have not been written or registered by a higher-strength `DataWriter`.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning Entity, this `QosPolicy` cannot be changed any more.

Attribute

The `QosPolicy` is controlled by the attribute kind which can be:

- `SHARED_OWNERSHIP_QOS` (default) - the same instance can be written by multiple `DataWriter` objects. All updates will be made available to the `DataReader` objects. In other words it does not have a specific owner
- `EXCLUSIVE_OWNERSHIP_QOS` - the instance will only be accepted from one `DataWriter` which is the only one whose modifications will be visible to the `DataReader` objects.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 9 Requested/Offered OwnershipQosPolicy

Requested Offered	SHARED	EXCLUSIVE
SHARED	compatible	INcompatible
EXCLUSIVE	INcompatible	compatible

Whether communication is established is controlled by the Data Distribution Service, depending on the Requested/Offered `QosPolicy` of the `DataWriter` and `DataReader`. The value of the `OWNERSHIP` kind offered must exactly match the one requested or else they are considered incompatible. As a consequence, a `DataWriter` that has an incompatible `QoS` with respect to what a `DataReader`

specified is not allowed to send its data to that specific `DataReader`. A `DataReader` that has an incompatible `QoS` with respect to what a `DataWriter` specified does not get any data from that particular `DataWriter`.

Exclusive ownership

The `DataWriter` with the highest `OwnershipStrengthQosPolicy` value and being alive (depending on the `LivelinessQosPolicy`) and which has not violated its `DeadlineQosPolicy` contract with respect to the instance, will be considered the owner of the instance. Consequently, the ownership can change as a result of:

- a `DataWriter` in the system with a higher value of the `OwnershipStrengthQosPolicy` modifies the instance
- a change in the `OwnershipStrengthQosPolicy` value (becomes less) of the `DataWriter` owning the instance
- a change in the liveliness (becomes not alive) of the `DataWriter` owning the instance
- a deadline with respect to the instance that is missed by the `DataWriter` that owns the instance.

Timeline

Each `DataReader` may detect the change of ownership at a different time. In other words, at a particular point in time, the `DataReader` objects do not have a consistent picture of who owns each instance for that `Topic`. Outside this grey area in time all `DataReader` objects will consider the same `DataWriter` to be the owner.

If multiple `DataWriter` objects with the same `OwnershipStrengthQosPolicy` modify the same instance, all `DataReader` objects will make the same choice of the particular `DataWriter` that is the owner. The `DataReader` is also notified of this via a status change that is accessible by means of the `Listener` or `Condition` mechanisms.

Ownership of an Instance

`DataWriter` objects are not aware whether they own a particular instance. There is no error or notification given to a `DataWriter` that modifies an instance it does not currently own.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this **qos** by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.12 OwnershipStrengthQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct OwnershipStrengthQosPolicy
{ Long value; };
```

Description

This QosPolicy specifies the value of the ownership strength of a DataWriter used to determine the ownership of an instance.

Attributes

Long value - specifies the ownership strength of the DataWriter.

Detailed Description

This QosPolicy specifies the value of the ownership strength of a DataWriter used to determine the ownership of an instance. This ownership is used to arbitrate among multiple DataWriter objects that attempt to modify the same instance. This QosPolicy only applies if the OwnershipQosPolicy is of kind EXCLUSIVE_OWNERSHIP_QOS. For more information, see OwnershipQosPolicy.

This QosPolicy is applicable to a DataWriter only. After enabling of the concerning Entity, this QosPolicy may be changed by using the `set_qos` operation. When changed, the ownership of the instances may change with it.

3.1.3.13 PartitionQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct PartitionQosPolicy
{ StringSeq name; };
```

Description

This QosPolicy specifies the logical partitions in which the Subscribers and Publishers are active.

Attributes

StringSeq name - holds the sequence of strings, which specifies the partitions.

Detailed Description

This `QosPolicy` specifies the logical partitions inside the domain in which the Subscribers and Publishers are active. This `QosPolicy` is particularly used to create a separate subspace, like a real domain versus a simulation domain. A Publisher and/or Subscriber can participate in more than one partition. Each string in the sequence of strings *name* defines a partition name. A partition name may contain wildcards. Sharing a partition means that at least one of the partition names in the sequence matches. When none of the partition names match, it is not considered an “incompatible” QoS and does not trigger any listeners or conditions. It only means, no communication is established. The default value of the attribute is an empty (zero-sized) sequence. This is treated as a special value that matches the “partition”.

This `QosPolicy` is applicable to a Publisher and Subscriber. After enabling of the concerning Entity, this `QosPolicy` may be changed by using the `set_qos` operation. When changed, it modifies the association of `DataReader` and `DataWriter` objects. It may establish new associations or break existing associations. By default, `DataWriter` and `DataReader` objects belonging to a Publisher or Subscriber that do not specify a `PartitionQosPolicy`, will participate in the default partition. In this case the partition name is “”.

Requested/Offered

The offered setting of this `QosPolicy` is independent of the one requested, in other words they are never considered incompatible. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side or `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side will not be raised

3.1.3.14 PresentationQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum PresentationQosPolicyAccessScopeKind
{ INSTANCE_PRESENTATION_QOS,
  TOPIC_PRESENTATION_QOS,
  GROUP_PRESENTATION_QOS };
```

```
struct PresentationQosPolicy
{
    PresentationQosPolicyAccessScopeKind access_scope;
    Boolean coherent_access;
    Boolean ordered_access;
};
```

Note: This QosPolicy is not yet implemented. It is scheduled for a future release.

Description

This QosPolicy controls how the samples representing changes to data instances are presented to the subscribing application.

Attributes

PresentationQosPolicyAccessScopeKind access_scope - specifies the samples controlled by this policy.

Boolean coherent_access - functionality behind this setting is not yet implemented. It is scheduled for a future release specifies whether the access should be coherent or not.

Boolean ordered_access - functionality behind this setting is not yet implemented. It is scheduled for a future release specifies whether the access should be ordered or not.

Detailed Description

This QosPolicy controls how the samples representing changes to data instances are presented to the subscribing application. In other words, how changes to data-instances can be made dependent on each other and also the kind of dependencies that can be propagated and maintained by the Data Distribution Service. It affects the application's ability to group a set of changes and to preserve the order in which they were sent. It also specifies the largest scope spanning the Entity objects for which the order and coherency of changes can be preserved.

This QosPolicy is applicable to a Publisher and Subscriber. After enabling of the concerning Entity, this QosPolicy cannot be changed any more.

Attributes

The two booleans control whether coherent access and ordered access are supported within the scope *access_scope*. Since only `INSTANCE_PRESENTATION_QOS` (the lowest level) is implemented, coherent access and ordered access are not applicable.

The changes to each instance are considered un-ordered relative to changes to any other instance. That means that changes (creations, deletions, modifications) made to two instances are not necessarily seen in the order they occur, the ordering applies

to each instance separately. This is the case even if the same application thread makes the changes using the same `DataWriter`. This is the default `access_scope`.

Note that this `QosPolicy` controls the ordering at which related changes are made available to the `Subscriber`. In other words the `Subscriber` can access the changes in the proper order. However, it does not necessarily imply that the `Subscriber` will indeed access the changes in the correct order. For that to occur, the application at the `Subscriber` end must use the proper logic in reading the `DataReader`.

Requested/Offered

In case the `Requested/Offered QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 10 Requested/Offered PresentationQosPolicy

<div>Requested Offered</div>	INSTANCE	Topic	Group
instance	compatible	INcompatible	INcompatible
topic	compatible	compatible	INcompatible
group	compatible	compatible	compatible

The default settings for this policy the only settings are currently supported. Deviations from the default setting will be ignored by the Publisher or Subscriber.

3.1.3.15 ReaderDataLifecycleQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct ReaderDataLifecycleQosPolicy
{
    Duration_t autopurge_nowriter_samples_delay;
    Duration_t autopurge_disposed_samples_delay; };
```

Description

This `QosPolicy` specifies the maximum duration for which the `DataReader` will maintain information regarding a data instance for which the `instance_state` becomes either `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`.

Attributes

Duration_t autopurge_nowriter_samples_delay - specifies the duration for which the `DataReader` will maintain information regarding a data instance for which the `instance_state` becomes `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`. By default the duration value is `DURATION_INFINITE`. When the delay time has expired, the data instance is marked so that it can be purged in the next garbage collection sweep.

Duration_t autopurge_disposed_samples_delay - specifies the duration for which the `DataReader` will maintain information regarding a data instance for which the `instance_state` becomes `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. By default the duration value is `DURATION_INFINITE`. When the delay time has expired, the data instance is marked so that it can be purged in the next garbage collection sweep.

Detailed Description

This `QosPolicy` specifies the maximum duration for which the `DataReader` will maintain information regarding a data instance for which the `instance_state` becomes either `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. The `DataReader` manages resources for instances and samples of those instances. The amount of resources managed depends on other `QosPolicies` like the `HistoryQosPolicy` and the `ResourceLimitsQosPolicy`. The `DataReader` can only release resources for data instances for which all samples have been taken and the `instance_state` has become `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. If an application does not take the samples belonging to a data instance with such an `instance_state`, the `DataReader` will never be able to release the maintained resources. By means of this `QosPolicy` the application can instruct the `DataReader` to release all resources related to the concerning data instance after a specified duration.

This `QosPolicy` is applicable to a `DataReader` only. After enabling of the concerning `DataReader`, this `QosPolicy` can be changed using the `set_qos` operation.

3.1.3.16 ReliabilityQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum ReliabilityQosPolicyKind
{ BEST_EFFORT_RELIABILITY_QOS,
```



```

        RELIABLE_RELIABILITY_QOS };
struct ReliabilityQosPolicy
{ ReliabilityQosPolicyKind kind;
  Duration_t max_blocking_time;
  Boolean synchronous; };

```

Description

This `QosPolicy` controls the level of reliability of the data distribution offered or requested by the `DataWriters` and `DataReaders`.

Attributes

ReliabilityQosPolicyKind kind - specifies the type of reliability which may be `BEST_EFFORT_RELIABILITY_QOS` or `RELIABLE_RELIABILITY_QOS`.

Duration_t max_blocking_time - specifies the maximum time the write operation may block when the `DataWriter` does not have space to store the value or when synchronous communication is specified and all expected acknowledgements are not yet received.

Boolean synchronous - specifies whether a `DataWriter` should wait for acknowledgements by all connected `DataReaders` that also have set a synchronous `ReliabilityQosPolicy`.

Detailed Description

This `QosPolicy` controls the level of reliability of the data distribution requested by a `DataReader` or offered by a `DataWriter`. In other words, it controls whether data is allowed to get lost in transmission or not.

This `QosPolicy` is applicable to a `DataReader`, `DataWriter` and `Topic`. After enabling of the concerning Entity, this `QosPolicy` cannot be changed any more.

Attributes

The `QosPolicy` is controlled by the attribute `kind` which can be:

- *RELIABLE_RELIABILITY_QOS* - the Data Distribution Service will attempt to deliver all samples in the `DataWriters` history; arrival-checks are performed and data may get re-transmitted in case of lost data. In the steady-state (no modifications communicated via the `DataWriter`) the Data Distribution Service guarantees that all samples in the `DataWriter` history will eventually be delivered to the all `DataReader` objects. Outside the steady-state the `HistoryQosPolicy` and `ResourceLimitsQosPolicy` determine how samples become part of the history and whether samples can be discarded from it. In this case also the `max_blocking_time` must be set

- *BEST_EFFORT_RELIABILITY_QOS* - the Data Distribution Service will only attempt to deliver the data; no arrival-checks are being performed and any lost data is not re-transmitted (non-reliable). Presumably new values for the samples are generated often enough by the application so that it is not necessary to resent or acknowledge any samples.

The effect of the attribute `max_blocking_time` depends on the setting of the `HistoryQosPolicy` and `ResourcesLimitsQosPolicy` and/or the synchronous setting of the `ReliabilityQosPolicy`. In case the `HistoryQosPolicy` kind is set to `KEEP_ALL_HISTORY_QOS`, the write operation on the `DataWriter` may block if the modification would cause one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. Also in case the synchronous attribute value of the `ReliabilityQosPolicy` is set to `TRUE` on both sides of a pair of connected `DataWriters` and `DataReaders`, then the `DataWriter` will wait until all its connected synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum duration the write operation may block.

Requested/Offered

In case the Requested/Offered `QosPolicy` are incompatible, the notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side and `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side is raised.

Table 11 Requested/Offered ReliabilityQosPolicy

Requested Offered	BEST_EFFORT	RELIABLE
BEST_EFFORT	compatible	Incompatible
RELIABLE	compatible	compatible

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.17 ResourceLimitsQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct ResourceLimitsQosPolicy
{
    Long max_samples;
    Long max_instances;
    Long max_samples_per_instance;
};
```

Note: This `QosPolicy` is not yet fully implemented. Missing features are scheduled for a future release.

Description

This `QosPolicy` will specify the maximum amount of resources, which can be used by a `DataWriter` or `DataReader`.

Attributes

Long max_samples - the maximum number of data samples for all instances for any single `DataWriter` (or `DataReader`). By default, `LENGTH_UNLIMITED`.

Long max_instances - the maximum number of instances for any single `DataWriter` (or `DataReader`). By default, `LENGTH_UNLIMITED`. Any other value than `LENGTH_UNLIMITED` will currently be ignored.

Long max_samples_per_instance - the maximum number of samples of any single instance for any single `DataWriter` (or `DataReader`). By default, `LENGTH_UNLIMITED`.

Detailed Description

This `QosPolicy` controls the maximum amount of resources that the Data Distribution Service can use in order to meet the requirements imposed by the application and other `QosPolicy` settings.

This `QosPolicy` is applicable to a `DataReader`, a `DataWriter` and a `Topic`. After enabling of the concerning Entity, this `QosPolicy` cannot be changed any more.

Requested/Offered

The value of the `QosPolicy` offered is independent of the one requested, in other words they are never considered incompatible. The communication will not be rejected on account of this `QosPolicy`. The notification `OFFERED_INCOMPATIBLE_QOS` status on the offering side or `REQUESTED_INCOMPATIBLE_QOS` status on the requesting side will not be raised.

Resource Limits

If the `DataWriter` objects are publishing samples faster than they are taken by the `DataReader` objects, the Data Distribution Service will eventually hit against some of the `QosPolicy`-imposed resource limits. Note that this may occur when just a single `DataReader` cannot keep up with its corresponding `DataWriter`.

In case the `HistoryQosPolicy` is `KEEP_LAST_HISTORY_QOS`, the setting of `ResourceLimitsQosPolicy` `max_samples_per_instance` must be compatible with the `HistoryQosPolicy` `depth`. For these two `QosPolicy` settings to be compatible, they must verify that `depth <= max_samples_per_instance`.

TopicQos

This `QosPolicy` can be set on a `Topic`. The `DataWriter` and/or `DataReader` can copy this qos by using the operations `copy_from_topic_qos` and then `set_qos`. That way the application can relatively easily ensure the `QosPolicy` for the `Topic`, `DataReader` and `DataWriter` are consistent.

3.1.3.18 SchedulingQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
enum SchedulingClassQosPolicyKind
{ SCHEDULE_DEFAULT,
  SCHEDULE_TIMESHARING,
  SCHEDULE_REALTIME };
struct SchedulingClassQosPolicy
{ SchedulingClassQosPolicyKind kind; };
enum SchedulingPriorityQosPolicyKind
{ PRIORITY_RELATIVE,
  PRIORITY_ABSOLUTE };
struct SchedulingPriorityQosPolicy
{ SchedulingPriorityQosPolicyKind kind; };
struct SchedulingQosPolicy
{ SchedulingClassQosPolicy scheduling_class;
  SchedulingPriorityQosPolicy scheduling_priority_kind;
  Long scheduling_priority; };
```

Description

This `QosPolicy` specifies the scheduling parameters that will be used for a thread that is spawned by the `DomainParticipant`.



Note that some scheduling parameters may not be supported by the underlying Operating System, or that you may need special privileges to select particular settings.

Attributes

SchedulingClassQosPolicyKind *scheduling_class.kind* - specifies the scheduling class used by the Operating System, which may be `SCHEDULE_DEFAULT`, `SCHEDULE_TIMESHARING` or `SCHEDULE_REALTIME`. Threads can only be spawned within the scheduling classes that are supported by the underlying Operating System.

SchedulingPriorityQosPolicyKind *scheduling_priority_kind.kind* - specifies the priority type, which may be either `PRIORITY_RELATIVE` or `PRIORITY_ABSOLUTE`.

Long scheduling_priority - specifies the priority that will be assigned to threads spawned by the `DomainParticipant`. Threads can only be spawned with priorities that are supported by the underlying Operating System.

Detailed Description

This `QosPolicy` specifies the scheduling parameters that will be used for threads spawned by the `DomainParticipant`. Note that some scheduling parameters may not be supported by the underlying Operating System, or that you may need special privileges to select particular settings. Refer to the documentation of your OS for more details on this subject.

Although the behaviour of the `scheduling_class` is highly dependent on the underlying OS, in general it can be said that when running in a `Timesharing` class your thread will have to yield execution to other threads of equal priority regularly. In a `Realtime` class your thread normally runs until completion, and can only be pre-empted by higher priority threads. Often the highest range of priorities is not accessible through a `Timesharing` Class.

The `scheduling_priority_kind` determines whether the specified `scheduling_priority` should be interpreted as an absolute priority, or whether it should be interpreted relative to the priority of its creator, in this case the priority of the thread that created the `DomainParticipant`.

3.1.3.19 TimeBasedFilterQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
```

```
struct TimeBasedFilterQosPolicy
{ Duration_t minimum_separation; };
```

Note: This `QosPolicy` is not yet implemented. It is scheduled for a future release.

3.1.3.20 TopicDataQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct TopicDataQosPolicy
{ OctetSeq value; };
```

Description

This `QosPolicy` allows the application to attach additional information to a `Topic` Entity. This information is distributed with the `BuiltinTopics`.

Attributes

OctetSeq value - a sequence of octets that holds the application topic data. By default, the sequence has length 0.

Detailed Description

This `QosPolicy` allows the application to attach additional information to a `Topic` Entity. This information is distributed with the `BuiltinTopic`. An application that discovers a new `Topic` entity, can use this information to add additional functionality. The `TopicDataQosPolicy` is changeable and updates of the `BuiltinTopic` instance must be expected. Note that the Data Distribution Service is not aware of the real structure of the topic data (the Data Distribution System handles it as an opaque type) and that the application is responsible for correct mapping on structural types for the specific platform.

3.1.3.21 TransportPriorityQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct TransportPriorityQosPolicy
{ Long value; };
```

Description

This `QosPolicy` specifies the priority with which the Data Distribution System can handle the data produced by the `DataWriter`.

Attributes

Long value - specifies the priority with which the Data Distribution System can handle the data produced by the `DataWriter`.

Detailed Description

This `QosPolicy` specifies the priority with which the Data Distribution System can handle the data produced by a `DataWriter`. This `QosPolicy` is considered to be a hint to the Data Distribution Service to control the priorities of the underlying transport means. A higher value represents a higher priority and the full range of the type is supported. By default the transport priority is set to 0.

The `TransportPriorityQosPolicy` is applicable to both `Topic` and `DataWriter` entities. After enabling of the concerning Entities, this `QosPolicy` may be changed by using the `set_qos` operation.

TopicQos

Note that changing this `QosPolicy` for the `Topic` does not influence the behaviour of the Data Distribution System for existing `DataWriter` entities because this `QosPolicy` is only used by the operation `copy_from_topic_qos` and when specifying `DATAWRITER_QOS_USE_TOPIC_QOS` when creating the `DataWriter`.

3.1.3.22 UserDataQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
    struct UserDataQosPolicy
    { OctetSeq value; };
```

Description

This `QosPolicy` allows the application to attach additional information to a `DomainParticipant`, `DataReader` or `DataWriter` entity. This information is distributed with the Builtin Topics.

Attributes

OctetSeq value - a sequence of octets that holds the application user data. By default, the sequence has length 0.

Detailed Description

This `QosPolicy` allows the application to attach additional information to a `DomainParticipant`, `DataReader` or `DataWriter` entity. This information is distributed with the Builtin Topics. An application that discovers a new Entity of the listed kind, can use this information to add additional functionality. The `UserDataQosPolicy` is changeable and updates of the Builtin Topic instance must be expected. Note that the Data Distribution Service is not aware of the real structure of the user data (the Data Distribution System handles it as an opaque type) and that the application is responsible for correct mapping on structural types for the specific platform.

3.1.3.23 WriterDataLifecycleQosPolicy

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct WriterDataLifecycleQosPolicy
{ Boolean autodispose_unregistered_instances; };
```

Note: The functionality behind this `QosPolicy` is not yet fully implemented. Missing features are scheduled for a future release.

Description

This `QosPolicy` specifies whether the Data Distribution Service should automatically dispose instances that are unregistered by the `DataWriter`.

Attributes

Boolean autodispose_unregistered_instances - specifies whether the Data Distribution Service should automatically dispose instances that are unregistered by this `DataWriter`.

Detailed Description

This `QosPolicy` controls the behaviour of the `DataWriter` with regards to the lifecycle of the data-instances it manages; that is, those data-instances that have been registered, either explicitly using one of the `register` operations, or implicitly by directly writing the data using the special `HANDLE_NIL` parameter. (See also Section 3.4.2.50, *register_instance*, on page 261).

The `autodispose_unregistered_instances` flag controls what happens when an instance gets unregistered by the `DataWriter`:

- If the `DataWriter` unregisters the instance explicitly using either `unregister_instance` or `unregister_instance_w_timestamp`, then the `autodispose_unregistered_instances` flag is currently ignored and the instance is never disposed automatically.
- If the `DataWriter` unregisters its instances implicitly because it is deleted or if a `DataReader` detects a loss of liveness of a connected `DataWriter`, then the `autodispose_unregistered_instances` flag determines whether the concerned instances are automatically disposed (`TRUE`) or not (`FALSE`).

The default value for the `autodispose_unregistered_instances` flag is `TRUE`. For `TRANSIENT` and `PERSISTENT` topics this means that all instances that are not explicitly unregistered by the application will by default be removed from the Transient and Persistent stores when the `DataWriter` is deleted or when a loss of its liveness is detected.

3.1.4 Listener Interface

This interface is the abstract base interface for all `Listener` interfaces. `Listeners` provide a generic mechanism for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. Each DCPS Entity supports its own specialized kind of `Listener`. `Listeners` are related to changes in communication status. For each Entity type, one specific `Listener` is derived from this interface. In the following modules, the following `Listeners` are derived from this interface:

- `DomainParticipantListener`
- `TopicListener`
- `PublisherListener`
- `DataWriterListener`
- `SubscriberListener`
- `DataReaderListener`

The Entity type specific `Listener` interfaces are part of the application which must implement the interface operations. A user defined class for these operations must be provided by the application which must extend from the **specific** `Listener` class. **All** `Listener` operations **must** be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

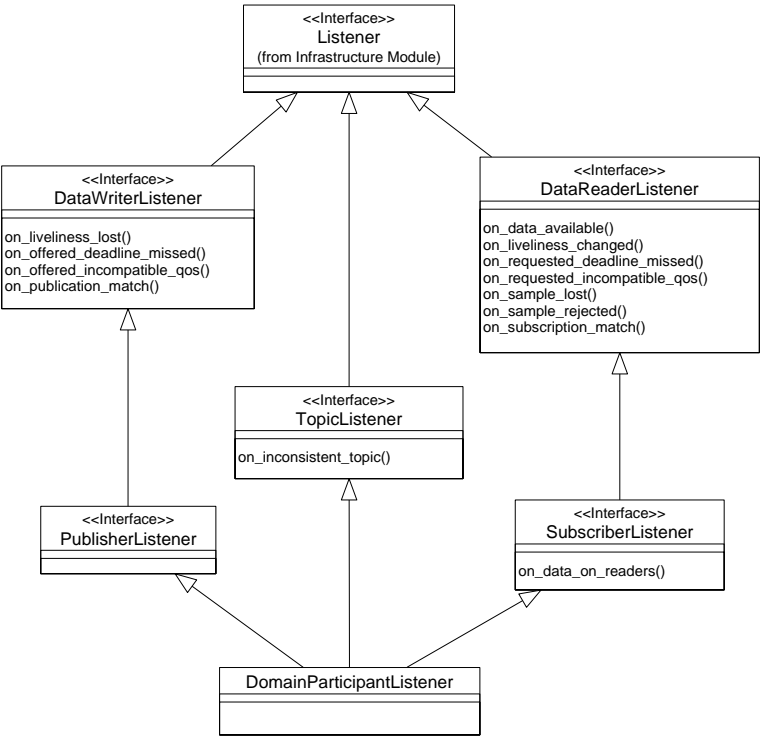


Figure 11 DCPS Listeners

The base class `Listener` does not contain any operations.

3.1.5 Struct Status

Each concrete `Entity` class has a set of `Status` attributes and for each attribute the `Entity` class provides an operation to read the value. Changes to `Status` attributes will affect associated `StatusCondition` and (invoked and associated) `Listener` objects.

The communication statuses whose changes can be communicated to the application depend on the `Entity`. The following table shows the relevant statuses for each `Entity`.

Table 12 Status Description Per Entity

Entity	Status	Meaning
Topic	INCONSISTENT_TOPIC_STATUS	Another <code>Topic</code> exists with the same name but with different characteristics.
Subscriber	DATA_ON_READERS_STATUS	New information is available.

Table 12 Status Description Per Entity

Entity	Status	Meaning
DataReader	SAMPLE_REJECTED_STATUS	A (received) sample has been rejected.
	LIVELINESS_CHANGED_STATUS	The liveliness of one or more DataWriter objects that were writing instances read through the DataReader has changed. Some DataWriter have become “alive” or “not alive”.
	REQUESTED_DEADLINE_MISSED_STATUS	The deadline that the DataReader was expecting through its DeadlineQosPolicy was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS_STATUS	A QosPolicy setting was incompatible with what is offered.
	DATA_AVAILABLE_STATUS	New information is available.
	SAMPLE_LOST_STATUS	A sample has been lost (never received).
	SUBSCRIPTION_MATCH_STATUS	The DataReader has found a DataWriter that matches the Topic and has compatible QoS.
DataWriter	LIVELINESS_LOST_STATUS	The liveliness that the DataWriter has committed through its LivelinessQosPolicy was not respected; thus DataReader objects will consider the DataWriter as no longer “alive”.
	OFFERED_DEADLINE_MISSED_STATUS	The deadline that the DataWriter has committed through its DeadlineQosPolicy was not respected for a specific instance.
	OFFERED_INCOMPATIBLE_QOS_STATUS	A QosPolicy setting was incompatible with what was requested.
	PUBLICATION_MATCH_STATUS	The DataWriter has found DataReader that matches the Topic and has compatible QoS.

A Status attribute can be retrieved with the operation `get_<status_name>_status`. For example, to get the `InconsistentTopicStatus` value, the application must call the operation `get_inconsistent_topic_status`.

Conceptually associated with each Entity communication status is a logical `StatusChangedFlag`. This flag indicates whether that particular communication status has changed. The `StatusChangedFlag` is only conceptual, therefore, it is not important whether this flag actually exists.

For the plain communication Status, the `StatusChangedFlag` is initially set to `FALSE`. It becomes `TRUE` whenever the plain communication Status changes and it is reset to `FALSE` each time the application accesses the plain communication Status via the proper `get_<status_name>_status` operation on the Entity.

A flag set means that a change has occurred since the last time the application has read its value.

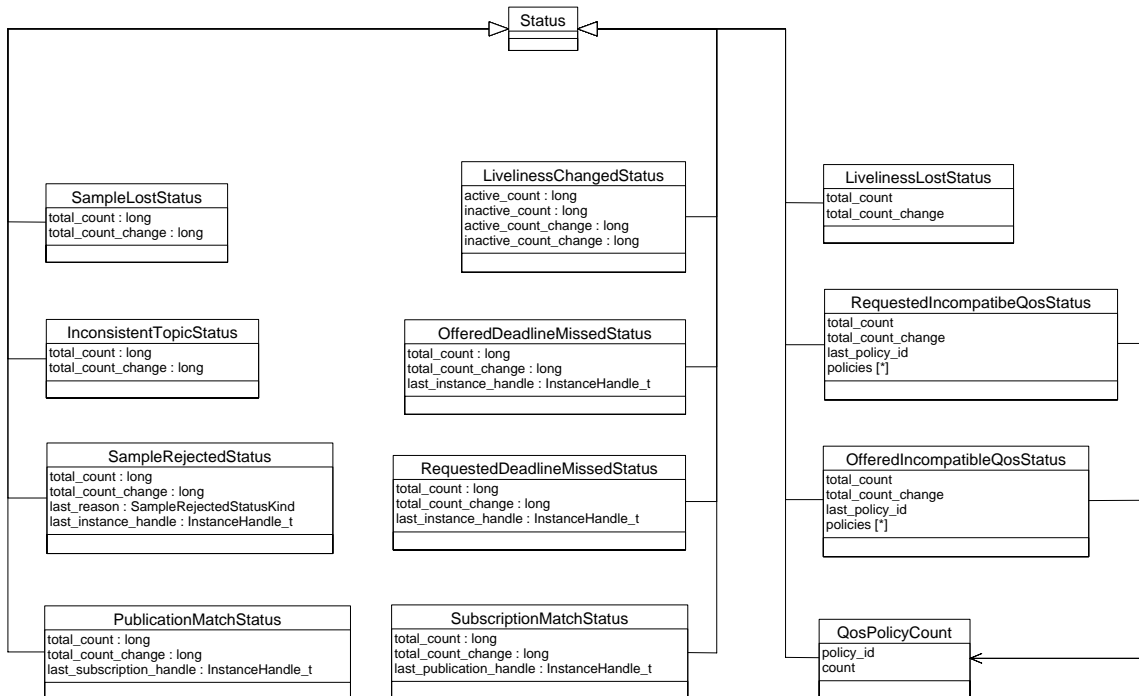


Figure 12 DCPS Status Values

Each Status attribute is implemented as a struct and therefore does not provide any operations. The interface description of these structs is as follows:

```

// struct <name>Status
//
struct InconsistentTopicStatus
{ Long total_count;
  Long total_count_change; };
struct SampleLostStatus
{ Long total_count;

```

```

        Long total_count_change; };
enum SampleRejectedStatusKind
{ NOT_REJECTED,
  REJECTED_BY_INSTANCES_LIMIT,
  REJECTED_BY_SAMPLES_LIMIT,
  REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT };
struct SampleRejectedStatus
{ Long total_count;
  Long total_count_change;
  SampleRejectedStatusKind last_reason;
  InstanceHandle_t last_instance_handle; };
struct LivelinessLostStatus
{ Long total_count;
  Long total_count_change; };
struct LivelinessChangedStatus
{ Long alive_count;
  Long not_alive_count;
  Long alive_count_change;
  Long not_alive_count_change;
  InstanceHandle_t last_publication_handle; };
struct OfferedDeadlineMissedStatus
{ Long total_count;
  Long total_count_change;
  InstanceHandle_t last_instance_handle; };
struct RequestedDeadlineMissedStatus
{ Long total_count;
  Long total_count_change;
  InstanceHandle_t last_instance_handle; };
struct OfferedIncompatibleQosStatus
{ Long total_count;
  Long total_count_change;
  QosPolicyId_t last_policy_id;
  QosPolicyCountSeq policies; };
struct RequestedIncompatibleQosStatus
{ Long total_count;
  Long total_count_change;
  QosPolicyId_t last_policy_id;
  QosPolicyCountSeq policies; };
struct PublicationMatchedStatus
{ Long total_count;
  Long total_count_change;
  Long current_count;
  Long current_count_change;
  InstanceHandle_t last_subscription_handle; };
struct SubscriptionMatchedStatus
{ Long total_count;
  Long total_count_change;
  Long current_count;
  Long current_count_change;
  InstanceHandle_t last_publication_handle; };

```

```
//
// implemented API operations
// <no operations>
```

The next paragraphs describe the usage of each <name>Status struct.

3.1.5.1 InconsistentTopicStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct InconsistentTopicStatus
{ Long total_count;
  Long total_count_change; };
```

Description

This struct contains the statistics about attempts to create other Topics with the same name but with different characteristics.

Attributes

Long total_count - the total detected cumulative count of Topic creations, whose name matches the Topic to which this Status is attached and whose characteristics are inconsistent.

Long total_count_change - the change in total_count since the last time the Listener was called or the Status was read.

Detailed Description

This struct contains the statistics about attempts to create other Topics with the same name but with different characteristics.

The attribute `total_count` holds the total detected cumulative count of Topic creations, whose name matches the Topic to which this Status is attached and whose characteristics are inconsistent.

The attribute `total_count_change` holds the incremental number of inconsistent Topics, since the last time the Listener was called or the Status was read.

3.1.5.2 LivelinessChangedStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct LivelinessChangedStatus
{
    Long alive_count;
    Long not_alive_count;
    Long alive_count_change;
    Long not_alive_count_change;
    InstanceHandle_t last_publication_handle; };
```

Description

This struct contains the statistics about whether the liveliness of one or more connected `DataWriter` objects has changed.

Attributes

Long alive_count - the total count of currently alive `DataWriter` objects that write the topic read by the `DataReader` to which this `Status` is attached.

Long not_alive_count - the total count of currently not alive `DataWriter` objects that wrote the topic read by the `DataReader` to which this `Status` is attached.

Long alive_count_change - the change in *alive_count* since the last time the `Listener` was called or the `Status` was read.

Long not_alive_count_change - the change in *not_alive_count* since the last time the `Listener` was called or the `Status` was read.

InstanceHandle_t last_publication_handle - handle to the last `DataWriter` whose change in liveliness caused this status to change.

Detailed Description

This struct contains the statistics about whether the liveliness of one or more connected `DataWriter` objects that were writing instances read through the `DataReader` has changed. In other words, some `DataWriter` have become “alive” or “not alive”.

The attribute *alive_count* holds the total number of currently alive `DataWriter` objects that write the topic read by the `DataReader` to which this `Status` is attached. This count increases when a newly-matched `DataWriter` asserts its liveliness for the first time or when a `DataWriter` previously considered to be not alive reasserts its liveliness. The count decreases when a `DataWriter` considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason.

The attribute *not_alive_count* holds the total count of currently not alive `DataWriters` that wrote the topic read by the `DataReader` to which this `Status` is attached, and that are no longer asserting their liveliness. This count increases

when a `DataWriter` considered alive fails to assert its liveliness and becomes not alive for some reason other than the normal deletion of that `DataWriter`. It decreases when a previously not alive `DataWriter` either reasserts its liveliness or is deleted normally.

The attribute `alive_count_change` holds the change in `alive_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `not_alive_count_change` holds the change in `not_alive_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_publication_handle` contains the instance handle to the `PublicationBuiltinTopicData` instance that represents the last `datawriter` whose change in liveliness caused this status to change. Be aware that this handle belongs to *another* `datareader`, the `PublicationBuiltinTopicDataDataReader` in the builtin-subscriber, and has no meaning in the context of the `datareader` from which the `LivelinessChangedStatus` was obtained. If the builtin-subscriber has not explicitly been obtained using `get_builtin_subscriber` on the `DomainParticipant`, then there is no `PublicationBuiltinTopicDataDataReader` as well, in which case the `last_publication_handle` will be set to `HANDLE_NIL`.



3.1.5.3 LivelinessLostStatus

Scope

DDS

Synopsis

```
#include <ccpp_dcps.h>
struct LivelinessLostStatus
{ Long total_count;
  Long total_count_change; };
```

Description

This struct contains the statistics about whether the liveliness of the `DataWriter` to which this `Status` is attached has been committed through its `LivelinessQosPolicy`.

Attributes

Long total_count - the total cumulative count of times the `DataWriter` to which this `Status` is attached failed to actively signal its liveliness within the offered liveliness period.

Long total_count_change - the change in *total_count* since the last time the Listener was called or the Status was read.

Detailed Description

This struct contains the statistics about whether the liveliness of the `DataWriter` to which this `Status` is attached has been committed through its `LivelinessQosPolicy`. In other words, whether the `DataWriter` failed to actively signal its liveliness within the offered liveliness period. In such a case, the connected `DataReader` objects will consider the `DataWriter` as no longer “alive”.

The attribute *total_count* holds the total cumulative number of times that the previously-alive `DataWriter` became not alive due to a failure to actively signal its liveliness within its offered liveliness period. This count does not change when an already not alive `DataWriter` simply remains not alive for another liveliness period.

The attribute *total_count_change* holds the change in *total_count* since the last time the Listener was called or the Status was read.

3.1.5.4 OfferedDeadlineMissedStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct OfferedDeadlineMissedStatus
{
    Long total_count
    Long total_count_change
    InstanceHandle_t last_instance_handle }

```

Description

This struct contains the statistics about whether the deadline that the `DataWriter` to which this `Status` is attached has committed through its `DeadlineQosPolicy` was not respected for a specific instance.

Attributes

Long total_count - the total cumulative count of times the `DataWriter` to which this `Status` is attached failed to write within its offered deadline.

Long total_count_change - the change in *total_count* since the last time the Listener was called or the Status was read.

InstanceHandle_t last_instance_handle - the handle to the last instance in the *DataWriter* to which this *Status* is attached, for which an offered deadline was missed.

Detailed Description

This struct contains the statistics about whether the deadline that the *DataWriter* to which this *Status* is attached has committed through its *DeadlineQosPolicy* was not respected for a specific instance.

The attribute *total_count* holds the total cumulative number of offered deadline periods elapsed during which the *DataWriter* to which this *Status* is attached failed to provide data. Missed deadlines accumulate; that is, each deadline period the *total_count* will be incremented by one.

The attribute *total_count_change* holds the change in *total_count* since the last time the *Listener* was called or the *Status* was read.

The attribute *last_instance_handle* holds the handle to the last instance in the *DataWriter* to which this *Status* is attached, for which an offered deadline was missed.

3.1.5.5 OfferedIncompatibleQosStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct OfferedIncompatibleQosStatus
{
    Long total_count
    Long total_count_change
    QosPolicyId_t last_policy_id
    QosPolicyCountSeq policies }

```

Description

This struct contains the statistics about whether an offered *QosPolicy* setting was incompatible with the requested *QosPolicy* setting.

Attributes

Long total_count - the total cumulative count of *DataReader* objects discovered by the *DataWriter* with the same *Topic* and *Partition* and with a requested *DataReaderQos* that was incompatible with the one offered by the *DataWriter*.

Long total_count_change - the change in *total_count* since the last time the *Listener* was called or the *Status* was read.

QosPolicyId_t last_policy_id - the id of one of the *QosPolicy* settings that was found to be incompatible with what was offered, the last time an incompatibility was detected.

QosPolicyCountSeq policies - a list containing for each *QosPolicy* the total number of times that the concerned *DataWriter* discovered a *DataReader* for the same *Topic* and a requested *DataReaderQos* that is incompatible with the one offered by the *DataWriter*.

Detailed Description

This struct contains the statistics about whether an offered *QosPolicy* setting was incompatible with the requested *QosPolicy* setting.

The Request/Offering mechanism is applicable between:

- the *DataWriter* and the *DataReader*. If the *QosPolicy* settings between *DataWriter* and *DataReader* are incompatible, no communication between them is established. In addition the *DataWriter* will be informed via a *REQUESTED_INCOMPATIBLE_QOS* status change and the *DataReader* will be informed via an *OFFERED_INCOMPATIBLE_QOS* status change.
- the *DataWriter* and the *Durability Service* (as a built-in *DataReader*). If the *QosPolicy* settings between *DataWriter* and the *Durability Service* are inconsistent, no communication between them is established. In that case data published by the *DataWriter* will not be maintained by the service and as a consequence will not be available for late joining *DataReaders*. The *QosPolicy* of the *Durability Service* in the role of *DataReader* is specified by the *DurabilityServiceQosPolicy* in the *Topic*.
- the *Durability Service* (as a built-in *DataWriter*) and the *DataReader*. If the *QosPolicy* settings between the *Durability Service* and the *DataReader* are inconsistent, no communication between them is established. In that case the *Durability Service* will not publish historical data to late joining *DataReaders*. The *QosPolicy* of the *Durability Service* in the role of *DataWriter* is specified by the *DurabilityServiceQosPolicy* in the *Topic*.

The attribute *total_count* holds the total cumulative count of *DataReader* objects discovered by the *DataWriter* with the same *Topic* and a requested *DataReaderQos* that was incompatible with the one offered by the *DataWriter*.

The attribute *total_count_change* holds the change in *total_count* since the last time the *Listener* was called or the *Status* was read.

The attribute *last_policy_id* holds the id of one of the *QosPolicy* settings that was found to be incompatible with what was offered, the last time an incompatibility was detected.

The attribute `policies` holds a list containing for each `QosPolicy` the total number of times that the concerned `DataWriter` discovered an incompatible `DataReader` for the same `Topic`. Each element in the list represents a counter for a different `QosPolicy`, identified by a corresponding unique index number. A named list of all index numbers is expressed as a set of constants in the API. See Table 13, *Overview of all named QosPolicy indexes* for an overview of all these constants.

Table 13 Overview of all named QosPolicy indexes

Index name	Index Value
INVALID_QOS_POLICY_ID	0
USERDATA_QOS_POLICY_ID	1
DURABILITY_QOS_POLICY_ID	2
PRESENTATION_QOS_POLICY_ID	3
DEADLINE_QOS_POLICY_ID	4
LATENCYBUDGET_QOS_POLICY_ID	5
OWNERSHIP_QOS_POLICY_ID	6
OWNERSHIPSTRENGTH_QOS_POLICY_ID	7
LIVELINESS_QOS_POLICY_ID	8
TIMEBASEDFILTER_QOS_POLICY_ID	9
PARTITION_QOS_POLICY_ID	10
RELIABILITY_QOS_POLICY_ID	11
DESTINATIONORDER_QOS_POLICY_ID	12
HISTORY_QOS_POLICY_ID	13
RESOURCELIMITS_QOS_POLICY_ID	14
ENTITYFACTORY_QOS_POLICY_ID	15
WRITERDATALIFECYCLE_QOS_POLICY_ID	16
READERDATALIFECYCLE_QOS_POLICY_ID	17
TOPICDATA_QOS_POLICY_ID	18
GROUPDATA_QOS_POLICY_ID	19
TRANSPORTPRIORITY_QOS_POLICY_ID	20
LIFESPAN_QOS_POLICY_ID	21
DURABILITYSERVICE_QOS_POLICY_ID	22

3.1.5.6 PublicationMatchedStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct PublicationMatchedStatus
{
    Long total_count
    Long total_count_change
    Long current_count;
    Long current_count_change;
    InstanceHandle_t last_subscription_handle }

```

Description

The functionality behind the `PublicationMatchedStatus` is not yet implemented. It is scheduled for a future release.

3.1.5.7 RequestedDeadlineMissedStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct RequestedDeadlineMissedStatus
{
    Long total_count
    Long total_count_change
    InstanceHandle_t last_instance_handle }

```

Description

This struct contains the statistics about whether the deadline that the `DataReader` to which this `Status` is attached was expecting through its `DeadlineQosPolicy`, was not respected for a specific instance.

Attributes

Long total_count - the total cumulative count of the missed deadlines detected for any instance read by the `DataReader` to which this `Status` is attached.

Long total_count_change - the change in *total_count* since the last time the `Listener` was called or the `Status` was read.

InstanceHandle_t last_instance_handle - the handle to the last instance in the `DataReader` to which this `Status` is attached for which a missed deadline was detected.

Detailed Description

This struct contains the statistics about whether the deadline that the `DataReader` to which this `Status` is attached was expecting through its `DeadlineQosPolicy` was not respected for a specific instance. Missed deadlines accumulate, that is, each deadline period the `total_count` will be incremented by one for each instance for which data was not received.

The attribute `total_count` holds the total cumulative count of the missed deadlines detected for any instance read by the `DataReader`.

The attribute `total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_instance_handle` holds the handle to the last instance in the `DataReader` for which a missed deadline was detected.

3.1.5.8 RequestedIncompatibleQosStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct RequestedIncompatibleQosStatus
{
    Long total_count
    Long total_count_change
    QosPolicyId_t last_policy_id
    QosPolicyCountSeq policies }

```

Description

This struct contains the statistics about whether a requested `QosPolicy` setting was incompatible with the offered `QosPolicy` setting.

Attributes

Long total_count - the total cumulative count of `DataWriter` objects, discovered by the `DataReader` to which this `Status` is attached, with the same `Topic` and an offered `DataWriterQos` that was incompatible with the one requested by the `DataReader`.

Long total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

QosPolicyId_t last_policy_id - the `<name>_QOS_POLICY_ID` of one of the `QosPolicies` that was found to be incompatible with what was requested, the last time an incompatibility was detected.

QosPolicyCountSeq policies - a list containing (for each QosPolicy) the total number of times that the concerned DataReader discovered a DataWriter with the same Topic and an offered DataWriterQos that is incompatible with the one requested by the DataReader.

Detailed Description

This struct contains the statistics about whether a requested QosPolicy setting was incompatible with the offered QosPolicy setting.

The Request/Offering mechanism is applicable between:

- the DataWriter and the DataReader. If the QosPolicy settings between DataWriter and DataReader are incompatible, no communication between them is established. In addition the DataWriter will be informed via a REQUESTED_INCOMPATIBLE_QOS status change and the DataReader will be informed via an OFFERED_INCOMPATIBLE_QOS status change.
- the DataWriter and the Durability Service (as a built-in DataReader). If the QosPolicy settings between DataWriter and the Durability Service are inconsistent, no communication between them is established. In that case data published by the DataWriter will not be maintained by the service and as a consequence will not be available for late joining DataReaders. The QosPolicy of the Durability Service in the role of DataReader is specified by the DurabilityServiceQosPolicy in the Topic.
- the Durability Service (as a built-in DataWriter) and the DataReader. If the QosPolicy settings between the Durability Service and the DataReader are inconsistent, no communication between them is established. In that case the Durability Service will not publish historical data to late joining DataReaders. The QosPolicy of the Durability Service in the role of DataWriter is specified by the DurabilityServiceQosPolicy in the Topic.

The attribute *total_count* holds the total cumulative count of DataWriter objects discovered by the DataReader with the same Topic and an offered DataWriterQos that was incompatible with the one requested by the DataReader.

The attribute *total_count_change* holds the change in *total_count* since the last time the Listener was called or the Status was read.

The attribute *last_policy_id* holds the *<name>_QOS_POLICY_ID* of one of the QosPolicies that was found to be incompatible with what was requested, the last time an incompatibility was detected.

The attribute *policies* holds a list containing for each QosPolicy the total number of times that the concerned DataReader discovered an incompatible DataWriter for the same Topic. Each element in the list represents a counter for a

different `QosPolicy`, identified by a corresponding unique index number. A named list of all index numbers is expressed as a set of constants in the API. See Table 13, *Overview of all named QosPolicy indexes*, on page 86 for an overview of all these constants.

3.1.5.9 SampleLostStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct SampleLostStatus
{
    Long total_count
    Long total_count_change }

```

Description

This struct contains the statistics about whether a sample has been lost (never received).

Attributes

Long total_count - the total cumulative count of all samples lost across all instances of data published under the `Topic`.

Long total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

Detailed Description

This struct contains the statistics about whether a sample has been lost (never received). The status is independent of the differences in instances, in other words, it includes all samples lost across all instances of data published under the `Topic`.

`total_count` holds the total cumulative count of all samples lost across all instances of data published under the `Topic`.

`total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

3.1.5.10 SampleRejectedStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>

```



```
enum SampleRejectedStatusKind
{
    NOT_REJECTED,
    REJECTED_BY_INSTANCES_LIMIT,
    REJECTED_BY_SAMPLES_LIMIT,
    REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT }

struct SampleRejectedStatus
{
    Long total_count
    Long total_count_change
    SampleRejectedStatusKind last_reason
    InstanceHandle_t last_instance_handle }
```

Description

This struct contains the statistics about samples that have been rejected.

Attributes

Long total_count - the total cumulative count of samples rejected by the `DataReader` to which this `Status` is attached.

Long total_count_change - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

SampleRejectedStatusKind last_reason - the reason for rejecting the last sample.

InstanceHandle_t last_instance_handle - the handle to the instance which would have been updated by the last sample that was rejected.

Detailed Description

This struct contains the statistics about whether a received sample has been rejected.

The attribute `total_count` holds the total cumulative count of samples rejected by the `DataReader` to which this `Status` is attached.

The attribute `total_count_change` holds the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

The attribute `last_reason` holds the reason for rejecting the last sample. The attribute can have the following values:

- *NOT_REJECTED* - no sample has been rejected yet.
- *REJECTED_BY_INSTANCES_LIMIT* - the sample was rejected because it would exceed the maximum number of instances set by the `ResourceLimitsQosPolicy`.
- *REJECTED_BY_SAMPLES_LIMIT* - the sample was rejected because it would exceed the maximum number of samples set by the `ResourceLimitsQosPolicy`.

- *REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT* - the sample was rejected because it would exceed the maximum number of samples per instance set by the ResourceLimitsQosPolicy.

The attribute `last_instance_handle` holds the handle to the instance which would have updated by the last sample that was rejected.

3.1.5.11 SubscriptionMatchedStatus

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct SubscriptionMatchedStatus
{
    Long total_count
    Long total_count_change
    Long current_count;
    Long current_count_change;
    InstanceHandle_t last_publication_handle }

```

Description

The functionality behind the `SubscriptionMatchedStatus` is not yet implemented. It is scheduled for a future release.

3.1.6 Class WaitSet

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

The `WaitSet` has no factory and must be created by the application. It is directly created as an object by using `WaitSet` constructors.

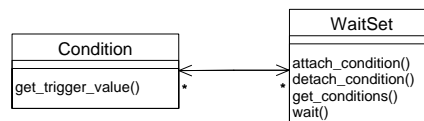


Figure 13 DCPS WaitSets

The interface description of this class is as follows:

```
class WaitSet
{
    //
    // implemented API operations
    //
    ReturnCode_t
    wait

```

```

        (ConditionSeq& active_conditions,
         const Duration_t& timeout);
    ReturnCode_t
        attach_condition
        (Condition_ptr cond);
    ReturnCode_t
        detach_condition
        (Condition_ptr cond);
    ReturnCode_t
        get_conditions
        (ConditionSeq& attached_conditions);
};

```

The following paragraphs describe the usage of all WaitSet operations.

3.1.6.1 attach_condition

Scope

DDS::WaitSet

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    attach_condition
    (Condition_ptr cond);

```

Description

This operation attaches a Condition to the WaitSet.

Parameters

in Condition_ptr cond - a pointer to a Condition.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation attaches a Condition to the WaitSet. The parameter *cond* must be either a ReadCondition, QueryCondition, StatusCondition or GuardCondition. To get this parameter see:

- ReadCondition created by `create_readcondition`
- QueryCondition created by `create_querycondition`
- StatusCondition retrieved by `get_statuscondition` on an Entity

- GuardCondition created by the C++ operation new.

When a GuardCondition is initially created, the `trigger_value` is FALSE.

When a Condition, whose `trigger_value` evaluates to TRUE, is attached to a WaitSet that is currently being waited on (using the `wait` operation), the WaitSet will unblock immediately.

Return Code

When the operation returns:

- `RETCODE_OK` - the Condition is attached to the WaitSet
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_BAD_PARAMETER` - the parameter `cond` is not a valid `Condition_ptr`
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.1.6.2 detach_condition

Scope

`DDS::WaitSet`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    detach_condition
        (Condition_ptr cond);
```

Description

This operation detaches a Condition from the WaitSet.

Parameters

in Condition_ptr cond - a pointer to a Condition in the WaitSet.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation detaches a Condition from the WaitSet. If the Condition was not attached to this WaitSet, the operation returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the Condition is detached from the WaitSet.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter *cond* is not a valid Condition_ptr.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the Condition was not attached to this WaitSet.

3.1.6.3 **get_conditions**

Scope

DDS::WaitSet

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_conditions
        (ConditionSeq_out attached_conditions);
```

Description

This operation retrieves the list of attached conditions.

Parameters

inout ConditionSeq& attached_conditions - a reference to a sequence which is used to pass the list of attached conditions.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation retrieves the list of attached conditions in the WaitSet. The parameter *attached_conditions* is a reference to a sequence which afterwards will refer to the sequence of attached conditions. The *attached_conditions* sequence and its buffer may be pre-allocated by the application and therefore must either be re-used in a subsequent invocation of the *get_conditions* operation or be released by invoking its destructor either implicitly or explicitly. If the pre-allocated sequence is not big enough to hold the number of triggered

Conditions, the sequence will automatically be (re-)allocated to fit the required size. The resulting sequence will either be an empty sequence, meaning there were no conditions attached, or will contain a list of `ReadCondition`, `QueryCondition`, `StatusCondition` and `GuardCondition`. These conditions previously have been attached by `attach_condition` and were created by there respective create operation:

- `ReadCondition` created by `create_readcondition`
- `QueryCondition` created by `create_querycondition`
- `StatusCondition` retrieved by `get_statuscondition` on an Entity
- `GuardCondition` created by the C++ operation `new`.

Return Code

When the operation returns:

- `RETCODE_OK` - the list of attached conditions is returned
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.1.6.4 wait

Scope

`DDS::WaitSet`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    wait
        (ConditionSeq& active_conditions,
         const Duration_t& timeout);
```

Description

This operation allows an application thread to wait for the occurrence of at least one of the conditions that is attached to the `WaitSet`.

Parameters

inout `ConditionSeq active_conditions` - a sequence which is used to pass the list of all the attached conditions that have a `trigger_value` of `TRUE`.

in const Duration_t& timeout - the maximum duration to block for the wait, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_TIMEOUT` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation allows an application thread to wait for the occurrence of at least one of the conditions to evaluate to `TRUE` that is attached to the `WaitSet`. If all of the conditions attached to the `WaitSet` have a `trigger_value` of `FALSE`, the wait operation will block the calling thread. The result of the operation is the continuation of the application thread after which the result is left in `active_conditions`. This is a reference to a sequence, which will contain the list of all the attached conditions that have a `trigger_value` of `TRUE`. The `active_conditions` sequence and its buffer may be pre-allocated by the application and therefore must either be re-used in a subsequent invocation of the wait operation or be released by invoking its destructor either implicitly or explicitly. If the pre-allocated sequence is not big enough to hold the number of triggered Conditions, the sequence will automatically be (re-)allocated to fit the required size. The parameter `timeout` specifies the maximum duration for the wait to block the calling application thread (when none of the attached conditions has a `trigger_value` of `TRUE`). In that case the return value is `RETCODE_TIMEOUT` and the `active_conditions` sequence is left empty. Since it is not allowed for more than one application thread to be waiting on the same `WaitSet`, the operation returns immediately with the value `RETCODE_PRECONDITION_NOT_MET` when the wait operation is invoked on a `WaitSet` which already has an application thread blocking on it.

Return Code

When the operation returns:

- `RETCODE_OK` - at least one of the attached conditions has a `trigger_value` of `TRUE`.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_TIMEOUT` - the timeout has elapsed without any of the attached conditions becoming `TRUE`.

- *RETCODE_PRECONDITION_NOT_MET* - the WaitSet already has an application thread blocking on it.

3.1.7 Class Condition

This class is the base class for all the conditions that may be attached to a WaitSet. This base class is specialized in three classes by the Data Distribution Service: GuardCondition, StatusCondition and ReadCondition (also there is a QueryCondition which is a specialized ReadCondition).

Each Condition has a *trigger_value* that can be TRUE or FALSE and is set by the Data Distribution Service (except a GuardCondition) depending on the evaluation of the Condition.

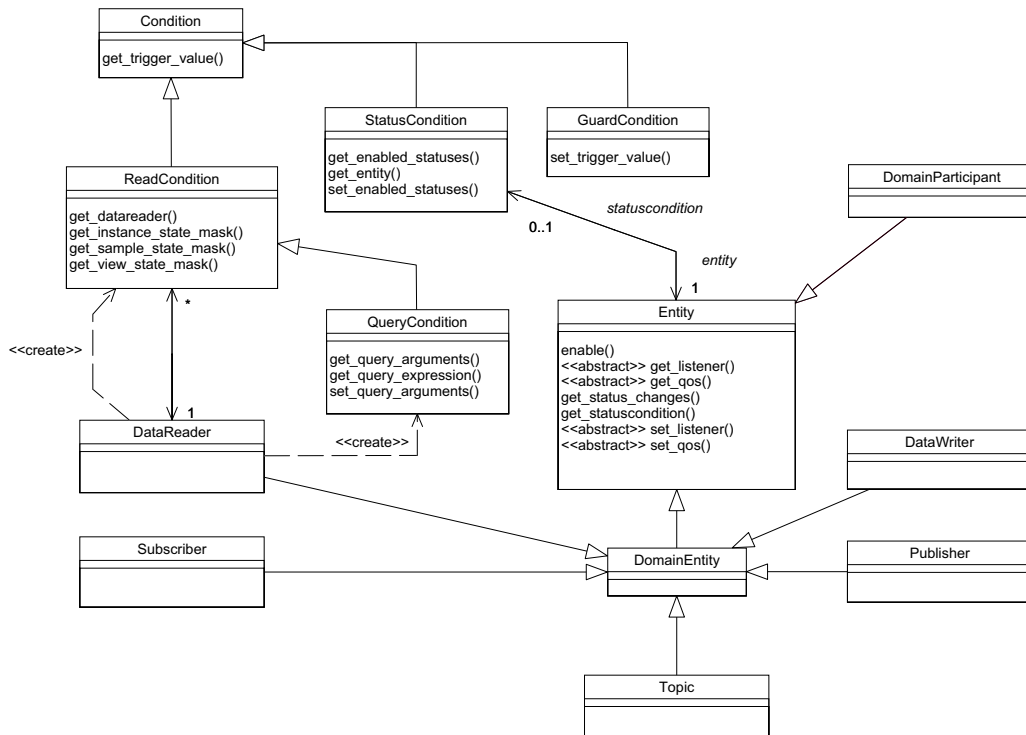


Figure 14 DCPS Conditions

The interface description of this class is as follows:

```

class Condition
{
    //
    // implemented API operations
    //
    Boolean
        get_trigger_value

```



```
        (void);
    };
```

The next paragraph describes the usage of the Condition operation.

3.1.7.1 `get_trigger_value`

Scope

DDS::Condition

Synopsis

```
#include <ccpp_dds_dcps.h>
Boolean
    get_trigger_value
        (void);
```

Description

This operation returns the `trigger_value` of the Condition.

Parameters

<none>

Return Value

Boolean - is the `trigger_value`.

Detailed Description

A Condition has a `trigger_value` that can be TRUE or FALSE and is set by the Data Distribution Service (except a GuardCondition). This operation returns the `trigger_value` of the Condition.

3.1.8 Class GuardCondition

A GuardCondition object is a specific Condition whose `trigger_value` is completely under the control of the application. The GuardCondition has no factory and must be created by the application. The GuardCondition is directly created as an object by using the GuardCondition constructor. When a GuardCondition is initially created, the `trigger_value` is FALSE. The purpose of the GuardCondition is to provide the means for an application to manually wake up a WaitSet. This is accomplished by attaching the GuardCondition to the Waitset and setting the `trigger_value` by means of the `set_trigger_value` operation.

The interface description of this class is as follows:

```
class GuardCondition
{
```

```

//
// inherited from Condition
//
// Boolean
//     get_trigger_value
//         (void);
//
// implemented API operations
//
//     ReturnCode_t
//         set_trigger_value
//             (Boolean value);
};

```

The next paragraphs describe the usage of all `GuardCondition` operations. The inherited operation is listed but not fully described since it is not implemented in this class. The full description of this operation is given in the class from which it is inherited. This is described in their respective paragraph.

3.1.8.1 `get_trigger_value` (inherited)

This operation is inherited and therefore not described here. See the class `Condition` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
Boolean
    get_trigger_value
        (void);

```

3.1.8.2 `set_trigger_value`

Scope

`DDS::GuardCondition`

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_trigger_value
        (Boolean value);

```

Description

This operation sets the `trigger_value` of the `GuardCondition`.

Parameters

in Boolean value - the boolean value to which the `GuardCondition` is set.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK or RETCODE_ERROR.

Detailed Description

A `GuardCondition` object is a specific `Condition` which `trigger_value` is completely under the control of the application. This operation must be used by the application to manually wake-up a `WaitSet`. This operation sets the `trigger_value` of the `GuardCondition` to the parameter value. The `GuardCondition` is directly created using the `GuardCondition` constructor. When a `GuardCondition` is initially created, the `trigger_value` is `FALSE`.

Return Code

When the operation returns:

- *RETCODE_OK* - the specified `trigger_value` has successfully been applied.
- *RETCODE_ERROR* - an internal error has occurred.

3.1.9 Class StatusCondition

Entity objects that have status attributes also have a `StatusCondition`, access is provided to the application by the `get_statuscondition` operation.

The communication statuses whose changes can be communicated to the application depend on the `Entity`. The following table shows the relevant statuses for each `Entity`.

Table 14 Status Per Entity

Entity	Status Name
Topic	INCONSISTENT_TOPIC_STATUS
Subscriber	DATA_ON_READERS_STATUS
DataReader	SAMPLE_REJECTED_STATUS
	LIVELINESS_CHANGED_STATUS
	REQUESTED_DEADLINE_MISSED_STATUS
	REQUESTED_INCOMPATIBLE_QOS_STATUS
	DATA_AVAILABLE_STATUS
	SAMPLE_LOST_STATUS
	SUBSCRIPTION_MATCHED_STATUS
DataWriter	LIVELINESS_LOST_STATUS
	OFFERED_DEADLINE_MISSED_STATUS
	OFFERED_INCOMPATIBLE_QOS_STATUS
	PUBLICATION_MATCHED_STATUS

The `trigger_value` of the `StatusCondition` depends on the communication statuses of that Entity (e.g., missed deadline) and also depends on the value of the `StatusCondition` attribute mask (`enabled_statuses` mask). A `StatusCondition` can be attached to a `WaitSet` in order to allow an application to suspend until the `trigger_value` has become `TRUE`.

The `trigger_value` of a `StatusCondition` will be `TRUE` if one of the enabled `StatusChangedFlags` is set. That is, `trigger_value==FALSE` only if all the values of the `StatusChangedFlags` are `FALSE`.

The sensitivity of the `StatusCondition` to a particular communication status is controlled by the list of `enabled_statuses` set on the condition by means of the `set_enabled_statuses` operation.

When the `enabled_statuses` are not changed by the `set_enabled_statuses` operation, all statuses are enabled by default.

The interface description of this class is as follows:

```
class StatusCondition
{
//
// inherited from Condition
//
// Boolean
//   get_trigger_value
//   (void);
//
// implemented API operations
//
    StatusMask
        get_enabled_statuses
        (void);

    ReturnCode_t
        set_enabled_statuses
        (StatusMask mask);

    Entity_ptr
        get_entity
        (void);
};
```

The next paragraphs describe the usage of all `StatusCondition` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.1.9.1 get_enabled_statuses

Scope

DDS::StatusCondition

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_enabled_statuses
        (void);
```

Description

This operation returns the list of enabled communication statuses of the StatusCondition.

Parameters

<none>

Return Value

StatusMask - a bit mask in which each bit shows which status is taken into account for the StatusCondition.

Detailed Description

The *trigger_value* of the StatusCondition depends on the communication status of that Entity (e.g., missed deadline, loss of information, etc.), ‘filtered’ by the set of *enabled_statuses* on the StatusCondition.

This operation returns the list of communication statuses that are taken into account to determine the *trigger_value* of the StatusCondition. This operation returns the statuses that were explicitly set on the last call to *set_enabled_statuses* or, if *set_enabled_statuses* was never called, the default list.

The result value is a bit mask in which each bit shows which status is taken into account for the StatusCondition. The relevant bits represents one of the following statuses:

- INCONSISTENT_TOPIC_STATUS
- OFFERED_DEADLINE_MISSED_STATUS
- REQUESTED_DEADLINE_MISSED_STATUS
- OFFERED_INCOMPATIBLE_QOS_STATUS
- REQUESTED_INCOMPATIBLE_QOS_STATUS
- SAMPLE_LOST_STATUS
- SAMPLE_REJECTED_STATUS

- DATA_ON_READERS_STATUS
- DATA_AVAILABLE_STATUS
- LIVELINESS_LOST_STATUS
- LIVELINESS_CHANGED_STATUS
- PUBLICATION_MATCHED_STATUS
- SUBSCRIPTION_MATCHED_STATUS

Each status bit is declared as a constant and can be used in an AND operation to check the status bit against the result of type `StatusMask`. Not all statuses are relevant to all `Entity` objects. See the respective `Listener` objects for each `Entity` for more information.

3.1.9.2 `get_entity`

Scope

`DDS::StatusCondition`

Synopsis

```
#include <ccpp_dds_dcps.h>
Entity_ptr
    get_entity
        (void);
```

Description

This operation returns the `Entity` associated with the `StatusCondition` or the `NULL` pointer.

Parameters

<none>

Return Value

Entity_ptr - a pointer to the `Entity` associated with the `StatusCondition` or the `NULL` pointer.

Detailed Description

This operation returns the `Entity` associated with the `StatusCondition`. Note that there is exactly one `Entity` associated with each `StatusCondition`. When the `Entity` was already deleted (there is no associated `Entity` any more), the `NULL` pointer is returned.

3.1.9.3 `get_trigger_value` (inherited)

This operation is inherited and therefore not described here. See the class `Condition` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
Boolean
    get_trigger_value
        (void);
```

3.1.9.4 set_enabled_statuses**Scope**

DDS::StatusCondition

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_enabled_statuses
        (StatusMask mask);
```

Description

This operation sets the list of communication statuses that are taken into account to determine the `trigger_value` of the `StatusCondition`.

Parameters

in `StatusMask mask` - a bit mask in which each bit sets the status which is taken into account for the `StatusCondition`.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_ALREADY_DELETED`.

Detailed Description

The `trigger_value` of the `StatusCondition` depends on the communication status of that Entity (e.g., missed deadline, loss of information, etc.), ‘filtered’ by the set of `enabled_statuses` on the `StatusCondition`.

This operation sets the list of communication statuses that are taken into account to determine the `trigger_value` of the `StatusCondition`. This operation may change the `trigger_value` of the `StatusCondition`.

`WaitSet` objects behaviour depend on the changes of the `trigger_value` of their attached `Conditions`. Therefore, any `WaitSet` to which the `StatusCondition` is attached is potentially affected by this operation.

If this function is not invoked, the default list of `enabled_statuses` includes all the statuses.

The parameter `mask` is a bit mask in which each bit shows which status is taken into account for the `StatusCondition`. The relevant bits represents one of the following states:

- `INCONSISTENT_TOPIC_STATUS`
- `OFFERED_DEADLINE_MISSED_STATUS`
- `REQUESTED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_ON_READERS_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_LOST_STATUS`
- `LIVELINESS_CHANGED_STATUS`
- `PUBLICATION_MATCHED_STATUS`
- `SUBSCRIPTION_MATCHED_STATUS`

Each status bit is declared as a constant and can be used in an OR operation to set the status bit in the parameter `mask` of type `StatusMask`. Not all statuses are relevant to all `Entity` objects. See the respective `Listener` objects for each `Entity` for more information.

Return Code

When the operation returns:

- `RETCODE_OK` - the list of communication statuses is set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `StatusCondition` has already been deleted.

- it acts as a factory for the Publisher, Subscriber, Topic, ContentFilteredTopic and MultiTopic objects
- it provides access to the built-in Topic objects
- it provides information about Topic objects
- It isolates applications within the same Domain (sharing the same domainId) from other applications in a different Domain on the same set of computers. In this way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.
- It provides administration services in the Domain, offering operations, which allow the application to ignore locally any information about a given Participant, Publication, Subscription or Topic.

The interface description of this class is as follows:

```
class DomainParticipant
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//   (void);
// StatusMask
//   get_status_changes
//   (void);
// ReturnCode_t
//   enable
//   (void);
//
// implemented API operations
//
    Publisher_ptr
        create_publisher
        (const PublisherQos& qos,
         PublisherListener_ptr a_listener,
         StatusMask mask);
    ReturnCode_t
        delete_publisher
        (Publisher_ptr p);
    Subscriber_ptr
        create_subscriber
        (const SubscriberQos& qos,
         SubscriberListener_ptr a_listener,
         StatusMask mask);
    ReturnCode_t
        delete_subscriber
        (Subscriber_ptr s);
```

```

Subscriber_ptr
    get_builtin_subscriber
        (void);
Topic_ptr
    create_topic
        (const char* topic_name,
         const char* type_name,
         const TopicQos& qos,
         TopicListener_ptr a_listener,
         StatusMask mask);
ReturnCode_t
    delete_topic
        (Topic_ptr a_topic);
Topic_ptr
    find_topic
        (const char* topic_name,
         const Duration_t& timeout);
TopicDescription_ptr
    lookup_topicdescription
        (const char* name);
ContentFilteredTopic_ptr
    create_contentfilteredtopic
        (const char* name,
         Topic_ptr related_topic,
         const char* filter_expression,
         const StringSeq& expression_parameters);
ReturnCode_t
    delete_contentfilteredtopic
        (ContentFilteredTopic_ptr
         a_contentfilteredtopic);
MultiTopic_ptr
    create_multitopic
        (const char* name,
         const char* type_name,
         const char* subscription_expression,
         const StringSeq& expression_parameters);
ReturnCode_t
    delete_multitopic
        (MultiTopic_ptr a_multitopic);
ReturnCode_t
    delete_contained_entities
        (void);
ReturnCode_t
    set_qos
        (const DomainParticipantQos& qos);
ReturnCode_t
    get_qos
        (DomainParticipantQos& qos);
ReturnCode_t
    set_listener

```

```

        (DomainParticipantListener_ptr a_listener,
         StatusMask mask);
DomainParticipantListener_ptr
    get_listener
        (void);
ReturnCode_t
    ignore_participant
        (InstanceHandle_t handle);
ReturnCode_t
    ignore_topic
        (InstanceHandle_t handle);
ReturnCode_t
    ignore_publication
        (InstanceHandle_t handle);
ReturnCode_t
    ignore_subscription
        (InstanceHandle_t handle);
DomainId_t
    get_domain_id
        (void);
ReturnCode_t
    get_discovered_participants
        (InstanceHandleSeq& participant_handles);
ReturnCode_t
    get_discovered_participant_data
        (ParticipantBuiltinTopicData& participant_data,
         InstanceHandle_t handle);
ReturnCode_t
    get_discovered_topics
        (InstanceHandleSeq& topic_handles);
ReturnCode_t
    get_discovered_topic_data
        (TopicBuiltinTopicData& topic_data,
         InstanceHandle_t handle);
ReturnCode_t
    assert_liveliness
        (void);
ReturnCode_t
    set_default_publisher_qos
        (const PublisherQos& qos);
ReturnCode_t
    get_default_publisher_qos
        (PublisherQos& qos);
ReturnCode_t
    set_default_subscriber_qos
        (const SubscriberQos& qos);
ReturnCode_t
    get_default_subscriber_qos
        (SubscriberQos& qos);
ReturnCode_t

```

```

        set_default_topic_qos
            (const TopicQos& qos);
    ReturnCode_t
        get_default_topic_qos
            (TopicQos& qos);
    Boolean
        contains_entity
            (InstanceHandle_t a_handle);
    ReturnCode_t
        get_current_time
            (Time_t& current_time);
};

```

The next paragraphs describe the usage of all DomainParticipant operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.2.1.1 assert_liveliness

Scope

DDS::DomainParticipant

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    assert_liveliness
        (void);

```

Description

This operation asserts the liveliness for the DomainParticipant.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or RETCODE_NOT_ENABLED.

Detailed Description

This operation will manually assert the liveliness for the DomainParticipant. This way, the Data Distribution Service is informed that the DomainParticipant is still alive. This operation only needs to be used when the DomainParticipant

`contains` DataWriters with the `LivelinessQosPolicy` set to `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS`, and it will only affect the liveliness of those DataWriters.

Writing data via the `write` operation of a `DataWriter` will assert the liveliness on the `DataWriter` itself and its `DomainParticipant`. Therefore, `assert_liveliness` is only needed when **not** writing regularly.

The liveliness should be asserted by the application, depending on the `LivelinessQosPolicy`.

Return Code

When the operation returns:

- `RETCODE_OK` - the liveliness of this `DomainParticipant` has successfully been asserted.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `DomainParticipant` is not enabled.

3.2.1.2 `contains_entity`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
Boolean
    contains_entity
        (InstanceHandle_t a_handle);
```

Description

This operation checks whether or not the given `Entity` represented by `a_handle` is created by the `DomainParticipant` or any of its contained entities.

Parameters

in `InstanceHandle_t a_handle` - an `Entity` in the Data Distribution System.

Return Value

Boolean - TRUE if `a_handle` represents an Entity that is created by the `DomainParticipant` or any of its contained Entities. Otherwise the return value is FALSE.

Detailed Description

This operation checks whether or not the given Entity represented by `a_handle` is created by the `DomainParticipant` itself (`TopicDescription`, `Publisher` or `Subscriber`) or created by any of its contained entities (`DataReader`, `ReadCondition`, `QueryCondition`, `DataWriter`, etc.).

Return value is TRUE if `a_handle` represents an Entity that is created by the `DomainParticipant` or any of its contained Entities. Otherwise the return value is FALSE.

3.2.1.3 `create_contentfilteredtopic`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ContentFilteredTopic_ptr
    create_contentfilteredtopic
        (const char* name,
         Topic_ptr related_topic,
         const char* filter_expression,
         const StringSeq& expression_parameters);
```

Description

This operation creates a `ContentFilteredTopic` for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a subset of the topic content.

Parameters

in const char name* - the name of the `ContentFilteredTopic`.

in Topic_ptr related_topic - the pointer to the base topic on which the filtering will be applied. Therefore, a filtered topic is based on an existing Topic.

in const char filter_expression* - the SQL expression (subset of SQL), which defines the filtering.

in const StringSeq& expression_parameters - the handle to a sequence of strings with the parameter value used in the SQL expression (i.e., the number of %n tokens in the expression). The number of values in

`expression_parameters` must be equal or greater than the highest referenced `%n` token in the `filter_expression` (e.g. if `%1` and `%8` are used as parameter in the `filter_expression`, the `expression_parameters` should at least contain `n+1 = 9` values).

Return Value

`ContentFilteredTopic_ptr` - the pointer to the newly created `ContentFilteredTopic`. In case of an error, a `NULL` pointer is returned.

Detailed Description

This operation creates a `ContentFilteredTopic` for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a subset of the topic content. The base topic, which is being filtered is defined by the parameter `related_topic`. The resulting `ContentFilteredTopic` only relates to the samples published under the `related_topic`, which have been filtered according to their content. The resulting `ContentFilteredTopic` only exists at the `DataReader` side and will never be published. The samples of the `related_topic` are filtered according to the SQL expression (which is a subset of SQL) as defined in the parameter `filter_expression` (see Appendix H, *DCPS Queries and Filters*).

The `filter_expression` may also contain parameters, which appear as `%n` tokens in the expression which must be set by the sequence of strings defined by the parameter `expression_parameters`. The number of values in `expression_parameters` must be equal or greater than the highest referenced `%n` token in the `filter_expression` (e.g. if `%1` and `%8` are used as parameter in the `filter_expression`, the `expression_parameters` should at least contain `n+1 = 9` values).

The `filter_expression` is a string that specifies the criteria to select the data samples of interest. In other words, it identifies the selection of data from the associated `Topics`. It is an SQL expression where the `WHERE` clause gives the content filter.

3.2.1.4 create_multitopic

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
MultiTopic_ptr
    create_multitopic
        (const char* name,
         const char* type_name,
```



```
const char* subscription_expression,
const StringSeq& expression_parameters);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation creates a `MultiTopic` for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a filtered/re-arranged combination and/or subset of the content of several topics.

Parameters

in const char name* - the name of the multi topic.

in const char type_name* - the name of the type of the `MultiTopic`. This `type_name` must have been registered using `register_type` prior to calling this operation.

in const char subscription_expression* - the SQL expression (subset of SQL), which defines the selection, filtering, combining and re-arranging of the sample data.

in const StringSeq& expression_parameters - the handle to a sequence of strings with the parameter value used in the SQL expression (i.e., the number of %n tokens in the expression). The number of values in `expression_parameters` must be equal or greater than the highest referenced %n token in the `subscription_expression` (e.g. if %1 and %8 are used as parameter in the `subscription_expression`, the `expression_parameters` should at least contain `n+1 = 9` values).

Return Value

MultiTopic_ptr - is the pointer to the newly created `MultiTopic`. In case of an error, a `NULL` pointer is returned.

Detailed Description

This operation creates a `MultiTopic` for a `DomainParticipant` in order to allow `DataReaders` to subscribe to a filtered/re-arranged combination and/or subset of the content of several topics. Before the `MultiTopic` can be created, the `type_name` of the `MultiTopic` must have been registered prior to calling this operation. Registering is done, using the `register_type` operation from `TypeSupport`. The list of topics and the logic, which defines the selection, filtering, combining and re-arranging of the sample data, is defined by the SQL expression (subset of SQL) defined in `subscription_expression`. The `subscription_expression` may also contain parameters, which appear as %n tokens in the expression. These parameters are defined in `expression_parameters`. The number of values in `expression_parameters`

must be equal or greater than the highest referenced %n token in the `subscription_expression` (e.g. if %1 and %8 are used as parameter in the `subscription_expression`, the `expression_parameters` should at least contain `n+1 = 9` values).

The `subscription_expression` is a string that specifies the criteria to select the data samples of interest. In other words, it identifies the selection and rearrangement of data from the associated `Topics`. It is an SQL expression where the `SELECT` clause provides the fields to be kept, the `FROM` part provides the names of the `Topics` that are searched for those fields, and the `WHERE` clause gives the content filter. The `Topics` combined may have different types but they are restricted in that the type of the fields used for the `NATURAL JOIN` operation must be the same.

The `DataReader`, which is associated with a `MultiTopic` only accesses information which exist locally in the `DataReader`, based on the `Topics` used in the `subscription_expression`. The actual `MultiTopic` will never be produced, only the individual `Topics`.

3.2.1.5 `create_publisher`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
Publisher_ptr
    create_publisher
        (const PublisherQos& qos,
         PublisherListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation creates a `Publisher` with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `PublisherListener` to it.

Parameters

in `const PublisherQos& qos` - a collection of `QosPolicy` settings for the new `Publisher`. In case these settings are not self consistent, no `Publisher` is created.

in `PublisherListener_ptr a_listener` - a pointer to the `PublisherListener` instance which will be attached to the new `Publisher`. It is permitted to use `NULL` as the value of the listener: this behaves as a `PublisherListener` whose operations perform no action.

in *StatusMask mask* - a bit-mask in which each bit enables the invocation of the *PublisherListener* for a certain status.

Return Value

Publisher_ptr - Return value is a pointer to the newly created *Publisher*. In case of an error, the *NULL* pointer is returned.

Detailed Description

This operation creates a *Publisher* with the desired *QosPolicy* settings and if applicable, attaches the optionally specified *PublisherListener* to it. When the *PublisherListener* is not applicable, the *NULL* pointer must be supplied instead. To delete the *Publisher* the operation *delete_publisher* or *delete_contained_entities* must be used.

In case the specified *QosPolicy* settings are not consistent, no *Publisher* is created and the *NULL* pointer is returned. The *NULL* pointer can also be returned when insufficient access rights exist for the partition(s) listed in the provided *QoS* structure.

Default QoS

The constant *PUBLISHER_QOS_DEFAULT* can be used as parameter *qos* to create a *Publisher* with the default *PublisherQos* as set in the *DomainParticipant*. The effect of using *PUBLISHER_QOS_DEFAULT* is the same as calling the operation *get_default_publisher_qos* and using the resulting *PublisherQos* to create the *Publisher*.

Communication Status

For each communication status, the *StatusChangedFlag* flag is initially set to *FALSE*. It becomes *TRUE* whenever that communication status changes. For each communication status activated in the mask, the associated *PublisherListener* operation 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 fact that the status is reset prior to calling the listener means that if the application calls the *get_<status_name>_status* from inside the listener it will see the status already reset.

The following statuses are applicable to the *PublisherListener*:

- *OFFERED_DEADLINE_MISSED_STATUS* *(propagated)*
- *OFFERED_INCOMPATIBLE_QOS_STATUS* *(propagated)*
- *LIVELINESS_LOST_STATUS* *(propagated)*
- *PUBLICATION_MATCHED_STATUS* *(propagated)*.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataWriterListener` of a contained `DataWriter`, the `DataWriterListener` on that contained `DataWriter` is invoked instead of the `PublisherListener`. This means that a status change on a contained `DataWriter` only invokes the `PublisherListener` if the contained `DataWriter` itself does not handle the trigger event generated by the status change.

In case a communication status is not activated in the mask of the `PublisherListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Publisher` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

3.2.1.6 create_subscriber

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
Subscriber_ptr
    create_subscriber
        (const SubscriberQos& qos,
         SubscriberListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation creates a `Subscriber` with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `SubscriberListener` to it.

Parameters

in const SubscriberQos& qos - a collection of QosPolicy settings for the new Subscriber. In case these settings are not self consistent, no Subscriber is created.

in SubscriberListener_ptr a_listener - a pointer to the SubscriberListener instance which will be attached to the new Subscriber. It is permitted to use NULL as the value of the listener: this behaves as a SubscriberListener whose operations perform no action.

in StatusMask mask - a bit-mask in which each bit enables the invocation of the SubscriberListener for a certain status.

Return Value

Subscriber_ptr - Return value is a pointer to the newly created Subscriber. In case of an error, the NULL pointer is returned.

Detailed Description

This operation creates a Subscriber with the desired QosPolicy settings and if applicable, attaches the optionally specified SubscriberListener to it. When the SubscriberListener is not applicable, the NULL pointer must be supplied instead. To delete the Subscriber the operation `delete_subscriber` or `delete_contained_entities` must be used.

In case the specified QosPolicy settings are not consistent, no Subscriber is created and the NULL pointer is returned. The NULL pointer can also be returned when insufficient access rights exist for the partition(s) listed in the provided QoS structure.

Default QoS

The constant `SUBSCRIBER_QOS_DEFAULT` can be used as parameter `qos` to create a Subscriber with the default SubscriberQos as set in the Domainparticipant. The effect of using `SUBSCRIBER_QOS_DEFAULT` is the same as calling the operation `get_default_subscriber_qos` and using the resulting SubscriberQos to create the Subscriber.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the `mask`, the associated SubscriberListener operation 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

fact that the status is reset prior to calling the listener means that if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `SubscriberListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS` *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `SAMPLE_LOST_STATUS` *(propagated)*
- `SAMPLE_REJECTED_STATUS` *(propagated)*
- `DATA_AVAILABLE_STATUS` *(propagated)*
- `LIVELINESS_CHANGED_STATUS` *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS` *(propagated)*.
- `DATA_ON_READERS_STATUS`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataReaderListener` of a contained `DataReader`, the `DataReaderListener` on that contained `DataReader` is invoked instead of the `SubscriberListener`. This means that a status change on a contained `DataReader` only invokes the `SubscriberListener` if the contained `DataReader` itself does not handle the trigger event generated by the status change.

In case a communication status is not activated in the `mask` of the `SubscriberListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Subscriber` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are “Read Communication Statuses” and are an exception to all other plain communication statuses: they have no corresponding status structure that can be

obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a `DataReader`, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

3.2.1.7 `create_topic`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
Topic_ptr
create_topic
    (const char* topic_name,
     const char* type_name,
     const TopicQos& qos,
     TopicListener_ptr a_listener,
     StatusMask mask);
```

Description

This operation creates a reference to a new or existing `Topic` under the given name, for a specific type, with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `TopicListener` to it.

Parameters

- in* `const char* topic_name` - the name of the `Topic` to be created. A new `Topic` will only be created, when no `Topic`, with the same name, is found within the `DomainParticipant`.
- in* `const char* type_name` - a local alias of the data type, which must have been registered before creating the `Topic`.
- in* `const TopicQos& qos` - a collection of `QosPolicy` settings for the new `Topic`. In case these settings are not self consistent, no `Topic` is created.
- in* `TopicListener_ptr a_listener` - a pointer to the `TopicListener` instance which will be attached to the new `Topic`. It is permitted to use `NULL` as the value of the listener: this behaves as a `TopicListener` whose operations perform no action.

in StatusMask mask - a bit-mask in which each bit enables the invocation of the `TopicListener` for a certain status.

Return Value

Topic_ptr - Return value is a pointer to the new or existing `Topic`. In case of an error, the `NULL` pointer is returned.

Detailed Description

This operation creates a reference to a new or existing `Topic` under the given name, for a specific type, with the desired `QosPolicy` settings and if applicable, attaches the optionally specified `TopicListener` to it. When the `TopicListener` is not applicable, the `NULL` pointer must be supplied instead. In case the specified `QosPolicy` settings are not consistent, no `Topic` is created and the `NULL` pointer is returned. To delete the `Topic` the operation `delete_topic` or `delete_contained_entities` must be used.

Default QoS

The constant `TOPIC_QOS_DEFAULT` can be used as parameter `qos` to create a `Topic` with the default `TopicQos` as set in the `DomainParticipant`. The effect of using `TOPIC_QOS_DEFAULT` is the same as calling the operation `get_default_topic_qos` and using the resulting `TopicQos` to create the `Topic`.

The `Topic` is bound to the type `type_name`. Prior to creating the `Topic`, the `type_name` must have been registered with the Data Distribution Service. Registering the `type_name` is done using the data type specific `register_type` operation.

Existing Topic Name

Before creating a new `Topic`, this operation performs a `lookup_topicdescription` for the specified `topic_name`. When a `Topic` is found with the same name in the current domain, the `QoS` and `type_name` of the found `Topic` are matched against the parameters `qos` and `type_name`. When they are the same, no `Topic` is created but a new proxy of the existing `Topic` is returned. When they are not exactly the same, no `Topic` is created and the `NULL` pointer is returned.

When a `Topic` is obtained multiple times, it must also be deleted that same number of times using `delete_topic` or calling `delete_contained_entities` once to delete all the proxies.

Local Proxy

Since a `Topic` is a global concept in the system, access is provided through a local proxy. In other words, the reference returned is actually not a reference to a `Topic` but to a locally created proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. For each create, a new proxy is created. Therefore the `Topic` must be deleted the same number of times, as the `Topic` was created with the same `topic_name` per Domain. In other words, each reference (local proxy) must be deleted separately.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the `mask`, the associated `TopicListener` operation 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 fact that the status is reset prior to calling the listener means that if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `TopicListener`:

- `INCONSISTENT_TOPIC_STATUS`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

In case a communication status is not activated in the `mask` of the `TopicListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Topic` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

3.2.1.8 delete_contained_entities

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_contained_entities
        (void);
```

Description

This operation deletes all the `Entity` objects that were created on the `DomainParticipant`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes all the `Entity` objects that were created on the `DomainParticipant`. In other words, it deletes all `Publisher`, `Subscriber`, `Topic`, `ContentFilteredTopic` and `MultiTopic` objects. Prior to deleting each contained `Entity`, this operation regressively calls the corresponding `delete_contained_entities` operation on each `Entity` (if applicable). In other words, all `Entity` objects in the `Publisher` and `Subscriber` are deleted, including the `DataWriter` and `DataReader`. Also the `QueryCondition` and `ReadCondition` objects contained by the `DataReader` are deleted.

Topic

Since a `Topic` is a global concept in the system, access is provided through a local proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. Such a proxy is created by the `create_topic` or `find_topic` operation. When a reference to the same `Topic` was created multiple times (either by `create_topic` or `find_topic`), all references (local proxies) are deleted. With the last proxy, the `Topic` itself is also removed from the system.



Note: The operation will return `PRECONDITION_NOT_MET` if the any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained `DataReader` cannot be deleted because the application has called a `read` or `take` operation and has not called the corresponding `return_loan` operation to return the loaned samples. In such cases, the operation does not roll-back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

- `RETCODE_OK` - the contained Entity objects are deleted and the application may delete the `DomainParticipant`.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - one or more of the contained entities are in a state where they cannot be deleted.

3.2.1.9 `delete_contentfilteredtopic`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_contentfilteredtopic
        (ContentFilteredTopic_ptr a_contentfilteredtopic);
```

Description

This operation deletes a `ContentFilteredTopic`.

Parameters

in `ContentFilteredTopic_ptr a_contentfilteredtopic` - a reference to the `ContentFilteredTopic`, which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes a `ContentFilteredTopic`.

The deletion of a `ContentFilteredTopic` is not allowed if there are any existing `DataReader` objects that are using the `ContentFilteredTopic`. If the `delete_contentfilteredtopic` operation is called on a `ContentFilteredTopic` with existing `DataReader` objects attached to it, it will return `PRECONDITION_NOT_MET`.

The `delete_contentfilteredtopic` operation must be called on the same `DomainParticipant` object used to create the `ContentFilteredTopic`. If `delete_contentfilteredtopic` is called on a different `DomainParticipant` the operation will have no effect and it will return `PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- `RETCODE_OK` - the `ContentFilteredTopic` is deleted
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_contentfilteredtopic` is not a valid `ContentFilteredTopic_ptr`
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - the operation is called on a different `DomainParticipant`, as used when the `ContentFilteredTopic` was created, or the `ContentFilteredTopic` is being used by one or more `DataReader` objects.

3.2.1.10 delete_multitopic

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_multitopic
        (MultiTopic_ptr a_multitopic);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation deletes a `MultiTopic`.

Parameters

in `MultiTopic_ptr a_multitopic` - a pointer to the `MultiTopic`, which is to be deleted.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes a `MultiTopic`.

The deletion of a `MultiTopic` is not allowed if there are any existing `DataReader` objects that are using the `MultiTopic`. If the `delete_multitopic` operation is called on a `MultiTopic` with existing `DataReader` objects attached to it, it will return `RETCODE_PRECONDITION_NOT_MET`.

The `delete_multitopic` operation must be called on the same `DomainParticipant` object used to create the `MultiTopic`. If `delete_multitopic` is called on a different `DomainParticipant` the operation will have no effect and it will return `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- `RETCODE_OK` - the `MultiTopic` is deleted
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_multitopic` is not a valid `MultiTopic_ptr`
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different DomainParticipant, as used when the MultiTopic was created, or the MultiTopic is being used by one or more DataReader objects.

3.2.1.11 delete_publisher

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_publisher
        (Publisher_ptr p);
```

Description

This operation deletes a Publisher.

Parameters

in Publisher_ptr p - a pointer to the Publisher, which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES* or *RETCODE_PRECONDITION_NOT_MET*.

Detailed Description

This operation deletes a Publisher. A Publisher cannot be deleted when it has any attached DataWriter objects. When the operation is called on a Publisher with DataWriter objects, the operation returns *RETCODE_PRECONDITION_NOT_MET*. When the operation is called on a different DomainParticipant, as used when the Publisher was created, the operation has no effect and returns *RETCODE_PRECONDITION_NOT_MET*.

Return Code

When the operation returns:

- *RETCODE_OK* - the Publisher is deleted
- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter *p* is not a valid *Publisher_ptr*
- *RETCODE_ALREADY_DELETED* - the *DomainParticipant* has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different *DomainParticipant*, as used when the *Publisher* was created, or the *Publisher* contains one or more *DataWriter* objects.

3.2.1.12 delete_subscriber

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_subscriber
        (Subscriber_ptr s);
```

Description

This operation deletes a *Subscriber*.

Parameters

in Subscriber_ptr s - a pointer to the *Subscriber*, which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES* or *RETCODE_PRECONDITION_NOT_MET*.

Detailed Description

This operation deletes a *Subscriber*. A *Subscriber* cannot be deleted when it has any attached *DataReader* objects. When the operation is called on a *Subscriber* with *DataReader* objects, the operation returns *RETCODE_PRECONDITION_NOT_MET*. When the operation is called on a different *DomainParticipant*, as used when the *Subscriber* was created, the operation has no effect and returns *RETCODE_PRECONDITION_NOT_MET*.

Return Code

When the operation returns:

- *RETCODE_OK* - the Subscriber is deleted
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter *s* is not a valid *Subscriber_ptr*
- *RETCODE_ALREADY_DELETED* - the *DomainParticipant* has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different *DomainParticipant*, as used when the Subscriber was created, or the Subscriber contains one or more *DataReader* objects.

3.2.1.13 delete_topic

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_topic
        (Topic_ptr a_topic);
```

Description

This operation deletes a *Topic*.

Parameters

in Topic_ptr a_topic - a pointer to the *Topic*, which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES* or *RETCODE_PRECONDITION_NOT_MET*.

Detailed Description

This operation deletes a *Topic*. A *Topic* cannot be deleted when there are any *DataReader*, *DataWriter*, *ContentFilteredTopic* or *MultiTopic* objects, which are using the *Topic*. When the operation is called on a *Topic* referenced by any of these objects, the operation returns *RETCODE_PRECONDITION_NOT_MET*.

When the operation is called on a different `DomainParticipant`, as used when the `Topic` was created, the operation has no effect and returns `RETCODE_PRECONDITION_NOT_MET`.

Local proxy

Since a `Topic` is a global concept in the system, access is provided through a local proxy. In other words, the reference is actually not a reference to a `Topic` but to the local proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. Such a proxy is created by the `create_topic` or `find_topic` operation. This operation will delete the local proxy. When a reference to the same `Topic` was created multiple times (either by `create_topic` or `find_topic`), each reference (local proxy) must be deleted separately. When this proxy is the last proxy for this `Topic`, the `Topic` itself is also removed from the system. As mentioned, a proxy may only be deleted when there are no other entities attached to it. However, it is possible to delete a proxy while there are entities attached to a different proxy.

Return Code

When the operation returns:

- `RETCODE_OK` - the `Topic` is deleted
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_topic` is not a valid `Topic_ptr`
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - the operation is called on a different `DomainParticipant`, as used when the `Topic` was created, or the `Topic` is still referenced by other objects.

3.2.1.14 **enable (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    enable
        (void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.15 find_topic

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
Topic_ptr
    find_topic
        (const char* topic_name,
         const Duration_t& timeout);
```

Description

This operation gives access to an existing (or ready to exist) enabled `Topic`, based on its `topic_name`.

Parameters

in const char topic_name* - the name of the `Topic` that the application wants access to.

in const Duration_t& timeout - the maximum duration to block for the `DomainParticipant_find_topic`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

Topic_ptr - Return value is a pointer to the `Topic` found.

Detailed Description

This operation gives access to an existing `Topic`, based on its `topic_name`. The operation takes as arguments the `topic_name` of the `Topic` and a timeout.

If a `Topic` of the same `topic_name` already exists, it gives access to this `Topic`. Otherwise it waits (blocks the caller) until another mechanism creates it. This other mechanism can be another thread, a configuration tool, or some other Data Distribution Service utility. If after the specified timeout the `Topic` can still not be found, the caller gets unblocked and the `NULL` pointer is returned.

A `Topic` obtained by means of `find_topic`, must also be deleted by means of `delete_topic` so that the local resources can be released. If a `Topic` is obtained multiple times it must also be deleted that same number of times using `delete_topic` or calling `delete_contained_entities` once to delete all the proxies.

A `Topic` that is obtained by means of `find_topic` in a specific `DomainParticipant` can only be used to create `DataReaders` and `DataWriters` in that `DomainParticipant` if its corresponding `TypeSupport` has been registered to that same `DomainParticipant`.

Local Proxy

Since a `Topic` is a global concept in the system, access is provided through a local proxy. In other words, the reference returned is actually not a reference to a `Topic` but to a locally created proxy. The Data Distribution Service propagates `Topics` and makes remotely created `Topics` locally available through this proxy. For each time this operation is called, a new proxy is created. Therefore the `Topic` must be deleted the same number of times, as the `Topic` was created with the same `topic_name` per `Domain`. In other words, each reference (local proxy) must be deleted separately.

3.2.1.16 `get_builtin_subscriber`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
Subscriber_ptr
    get_builtin_subscriber
        (void);
```

Description

This operation returns the built-in `Subscriber` associated with the `DomainParticipant`.

Parameters

<none>

Return Value

`Subscriber_ptr` - Result value is a pointer to the built-in `Subscriber` associated with the `DomainParticipant`.

Detailed Description

This operation returns the built-in `Subscriber` associated with the `DomainParticipant`. Each `DomainParticipant` contains several built-in `Topic` objects. The built-in `Subscriber` contains the corresponding `DataReader`

objects to access them. All these `DataReader` objects belong to a single built-in `Subscriber`. Note that there is exactly one built-in `Subscriber` associated with each `DomainParticipant`.

3.2.1.17 `get_current_time`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_current_time
        (Time_t& current_time);
```

Description

This operation returns the value of the current time that the Data Distribution Service uses to time-stamp written data as well as received data in `current_time`.

Parameters

inout `Time_t& current_time` - the value of the current time as used by the Data Distribution System. The input value of `current_time` is ignored.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation returns the value of the current time that the Data Distribution Service uses to time-stamp written data as well as received data in `current_time`. The input value of `current_time` is ignored by the operation.

Return Code

When the operation returns:

- `RETCODE_OK` - the value of the current time is returned in `current_time`.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `current_time` is not a valid reference.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the DomainParticipant is not enabled.

3.2.1.18 `get_default_publisher_qos`

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_default_publisher_qos
        (PublisherQos& qos);
```

Description

This operation gets the struct with the default Publisher QosPolicy settings of the DomainParticipant.

Parameters

inout PublisherQos& qos - a reference to the PublisherQos struct (provided by the application) in which the default QosPolicy settings for the Publisher are written.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_ALREADY_DELETED* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation gets the struct with the default Publisher QosPolicy settings of the DomainParticipant (that is the PublisherQos) which is used for newly created Publisher objects, in case the constant *PUBLISHER_QOS_DEFAULT* is used. The default PublisherQos is only used when the constant is supplied as parameter *qos* to specify the PublisherQos in the *create_publisher* operation. The application must provide the PublisherQos struct in which the QosPolicy settings can be stored and pass the *qos* reference to the operation. The operation writes the default QosPolicy settings to the struct referenced to by *qos*. Any settings in the struct are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_publisher_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` setting as defined in Table 2 on page 37

Return Code

When the operation returns:

- `RETCODE_OK` - the default Publisher `QosPolicy` settings of this `DomainParticipant` have successfully been copied into the specified `PublisherQos` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.19 `get_default_subscriber_qos`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_default_subscriber_qos
        (SubscriberQos& qos);
```

Description

This operation gets the struct with the default `Subscriber QosPolicy` settings of the `DomainParticipant`.

Parameters

inout `SubscriberQos& qos` - a reference to the `QosPolicy` struct (provided by the application) in which the default `QosPolicy` settings for the `Subscriber` is written.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets the struct with the default `Subscriber QoS Policy` settings of the `DomainParticipant` (that is the `SubscriberQos`) which is used for newly created `Subscriber` objects, in case the constant `SUBSCRIBER_QOS_DEFAULT` is used. The default `SubscriberQos` is only used when the constant is supplied as parameter `qos` to specify the `SubscriberQos` in the `create_subscriber` operation. The application must provide the `QoS` struct in which the policy can be stored and pass the `qos` reference to the operation. The operation writes the default `QoS Policy` to the struct referenced to by `qos`. Any settings in the struct are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_subscriber_qos`, or, if the call was never made, the default values as specified for each `QoS Policy` defined in Table 2 on page 37

Return Code

When the operation returns:

- `RETCODE_OK` - the default `Subscriber QoS Policy` settings of this `DomainParticipant` have successfully been copied into the specified `SubscriberQos` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.20 `get_default_topic_qos`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_default_topic_qos
        (TopicQos& qos);
```

Description

This operation gets the struct with the default `Topic QoS Policy` settings of the `DomainParticipant`.

Parameters

inout TopicQos& qos - a reference to the `QosPolicy` struct (provided by the application) in which the default `QosPolicy` settings for the `Topic` is written.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets the struct with the default `Topic QosPolicy` settings of the `DomainParticipant` (that is the `TopicQos`) which is used for newly created `Topic` objects, in case the constant `TOPIC_QOS_DEFAULT` is used. The default `TopicQos` is only used when the constant is supplied as parameter `qos` to specify the `TopicQos` in the `create_topic` operation. The application must provide the `QoS` struct in which the policy can be stored and pass the `qos` reference to the operation. The operation writes the default `QosPolicy` to the struct referenced to by `qos`. Any settings in the struct are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_topic_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` defined in Table 2 on page 37

Return Code

When the operation returns:

- `RETCODE_OK` - the default `Topic QosPolicy` settings of this `DomainParticipant` have successfully been copied into the specified `TopicQos` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.21 `get_discovered_participants`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
```



```

    ReturnCode_t
    get_discovered_participants
    (InstanceHandleSeq& participant_handles);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.22 `get_discovered_participant_data`

Scope

DDS::DomainParticipant

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_discovered_participant_data
    (ParticipantBuiltinTopicData& participant_data,
     InstanceHandle_t handle);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.23 `get_discovered_topics`

Scope

DDS::DomainParticipant

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_discovered_topics
    (InstanceHandleSeq& topic_handles);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.24 `get_discovered_topic_data`

Scope

DDS::DomainParticipant

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_discovered_topic_data
    (TopicBuiltinTopicData& topic_data,
     InstanceHandle_t handle);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.25 `get_domain_id`

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainId_t
    get_domain_id
        (void);
```

Description

This operation returns the `DomainId` of the Domain to which this `DomainParticipant` is attached.

Parameters

<none>

Return Value

DomainId_t - result is the `DomainId`.

Detailed Description

This operation returns the `DomainId` of the Domain to which this `DomainParticipant` is attached. A `DomainId` consists of a string that represents either a URI to the location of the configuration file (e.g. "file:///projects/DDS/ospl.xml") or the Domain name as specified in the configuration file. The actual value returned is dependent of the value used when creating the `DomainParticipant`, also see the `DomainParticipantFactory` `create_participant` operation. If a `DomainParticipant` is created using the Domain name then it will also return the Domain name, and *vice versa*: when created using a URI then the URI will be returned by this operation. The configuration file, identified by the URI, specifies all configuration details of the Domain.

A `DomainId` may contain the `NULL` pointer: in that case the location of the configuration file is extracted from the environment variable called `OSPL_URI`.

3.2.1.26 `get_listener`

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipantListener_ptr
    get_listener
        (void);
```

Description

This operation allows access to a DomainParticipantListener.

Parameters

<none>

Return Value

DomainParticipantListener_ptr - result is a pointer to the DomainParticipantListener attached to the DomainParticipant.

Detailed Description

This operation allows access to a DomainParticipantListener attached to the DomainParticipant. When no DomainParticipantListener was attached to the DomainParticipant, the NULL pointer is returned.

3.2.1.27 get_qos**Scope**

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (DomainParticipantQos& qos);
```

Description

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

Parameters

inout DomainParticipantQos& qos - a reference to the destination DomainParticipantQos struct in which the QosPolicy settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `DomainParticipant` on which this operation is used. This `DomainParticipantQos` is stored at the location pointed to by the `qos` parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this `DomainParticipant` has successfully been copied into the specified `DomainParticipantQos` parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the `DomainParticipant` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.28 `get_status_changes` (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_status_changes
        (void);
```

3.2.1.29 `get_statuscondition` (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition
    get_statuscondition_ptr
        (void);
```

3.2.1.30 ignore_participant

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    ignore_participant
        (InstanceHandle_t handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.31 ignore_publication

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    ignore_publication
        (InstanceHandle_t handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.32 ignore_subscription

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    ignore_subscription
        (InstanceHandle_t handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.33 ignore_topic

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
```

```
ignore_topic
    (InstanceHandle_t handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.2.1.34 lookup_topicdescription

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
TopicDescription_ptr
    lookup_topicdescription
        (const char* name);
```

Description

This operation gives access to a locally-created TopicDescription, with a matching name.

Parameters

in const char name* - the name of the TopicDescription to look for.

Return Value

TopicDescription_ptr - Return value is a pointer to the TopicDescription found. When no such TopicDescription is found, the NULL pointer is returned.

Detailed Description

The operation `lookup_topicdescription` gives access to a locally-created TopicDescription, based on its name. The operation takes as argument the name of the TopicDescription.

If one or more local TopicDescription proxies (see also section 3.2.1.15) of the same name already exist, a pointer to one of the already existing local proxies is returned: `lookup_topicdescription` will never create a new local proxy. That means that the proxy that is returned does not need to be deleted separately from its original. When no local proxy exists, it returns the NULL pointer. The operation never blocks.

The operation `lookup_topicdescription` may be used to locate any locally-created Topic, ContentFilteredTopic and MultiTopic object.

3.2.1.35 `set_default_publisher_qos`

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_default_publisher_qos
        (const PublisherQos& qos);
```

Description

This operation sets the default `PublisherQos` of the `DomainParticipant`.

Parameters

in `const PublisherQos& qos` - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created Publishers.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation sets the default `PublisherQos` of the `DomainParticipant` (that is the struct with the `QosPolicy` settings) which is used for newly created Publisher objects, in case the constant `PUBLISHER_QOS_DEFAULT` is used. The default `PublisherQos` is only used when the constant is supplied as parameter `qos` to specify the `PublisherQos` in the `create_publisher` operation. The `PublisherQos` is always self consistent, because its policies do not depend on each other. This means this operation never returns the `RETCODE_INCONSISTENT_POLICY`. The values set by this operation are returned by `get_default_publisher_qos`.

Return Code

When the operation returns:

- `RETCODE_OK` - the new default `PublisherQos` is set
- `RETCODE_ERROR` - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter *qos* is not a valid *PublisherQos*. It contains a *QosPolicy* setting with an enum value that is outside its legal boundaries.
- *RETCODE_UNSUPPORTED* - one or more of the selected *QosPolicy* values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the *DomainParticipant* has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.36 **set_default_subscriber_qos**

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_default_subscriber_qos
        (const SubscriberQos& qos);
```

Description

This operation sets the default *SubscriberQos* of the *DomainParticipant*.

Parameters

in const SubscriberQos& qos - a collection of *QosPolicy* settings, which contains the new default *QosPolicy* settings for the newly created *Subscribers*.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_UNSUPPORTED*, *RETCODE_ALREADY_DELETED* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation sets the default *SubscriberQos* of the *DomainParticipant* (that is the struct with the *QosPolicy* settings) which is used for newly created *Subscriber* objects, in case the constant *SUBSCRIBER_QOS_DEFAULT* is used. The default *SubscriberQos* is only used when the constant is supplied as parameter *qos* to specify the *SubscriberQos* in the *create_subscriber* operation. The *SubscriberQos* is always self consistent, because its policies do

not depend on each other. This means this operation never returns the `RETCODE_INCONSISTENT_POLICY`. The values set by this operation are returned by `get_default_subscriber_qos`.

Return Code

When the operation returns:

- `RETCODE_OK` - the new default `SubscriberQos` is set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `qos` is not a valid `PublisherQos`. It contains a `QosPolicy` setting with an enum value that is outside its legal boundaries.
- `RETCODE_UNSUPPORTED` - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- `RETCODE_ALREADY_DELETED` - the `DomainParticipant` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.37 `set_default_topic_qos`

Scope

`DDS::DomainParticipant`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_default_topic_qos
        (const TopicQos& qos);
```

Description

This operation sets the default `TopicQos` of the `DomainParticipant`.

Parameters

in `const TopicQos& qos` - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created Topics.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_INCONSISTENT_POLICY`.

Detailed Description

This operation sets the default TopicQos of the DomainParticipant (that is the struct with the QosPolicy settings) which is used for newly created Topic objects, in case the constant TOPIC_QOS_DEFAULT is used. The default TopicQos is only used when the constant is supplied as parameter qos to specify the TopicQos in the create_topic operation. This operation checks if the TopicQos is self consistent. If it is not, the operation has no effect and returns RETCODE_INCONSISTENT_POLICY. The values set by this operation are returned by get_default_topic_qos.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default TopicQos is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid TopicQos. It contains a QosPolicy setting with an invalid Duration_t value or an enum value that is outside its legal boundaries
- *RETCODE_UNSUPPORTED* - one or more of the selected QosPolicy values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the DomainParticipant has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_INCONSISTENT_POLICY* - the parameter qos contains conflicting QosPolicy settings, e.g. a history depth that is higher than the specified resource limits.

3.2.1.38 set_listener

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_listener
        (DomainParticipantListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation attaches a `DomainParticipantListener` to the `DomainParticipant`.

Parameters

in `DomainParticipantListener_ptr a_listener` - a pointer to the `DomainParticipantListener` instance, which will be attached to the `DomainParticipant`.

in `StatusMask mask` - a bit mask in which each bit enables the invocation of the `DomainParticipantListener` for a certain status.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation attaches a `DomainParticipantListener` to the `DomainParticipant`. Only one `DomainParticipantListener` can be attached to each `DomainParticipant`. If a `DomainParticipantListener` was already attached, the operation will replace it with the new one. When `a_listener` is the `NULL` pointer, it represents a listener that is treated as a NOOP¹ for all statuses activated in the bit mask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `DomainParticipantListener` operation 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>_status` from inside the listener it will see the status already reset. An exception to this rule is the `NULL` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `DomainParticipantListener`:

- `INCONSISTENT_TOPIC_STATUS` *(propagated)*
- `OFFERED_DEADLINE_MISSED_STATUS` *(propagated)*
- `REQUESTED_DEADLINE_MISSED_STATUS` *(propagated)*

1. Short for **No-Operation**, an instruction that does nothing.

- OFFERED_INCOMPATIBLE_QOS_STATUS (propagated)
- REQUESTED_INCOMPATIBLE_QOS_STATUS (propagated)
- SAMPLE_LOST_STATUS (propagated)
- SAMPLE_REJECTED_STATUS (propagated)
- DATA_ON_READERS_STATUS (propagated)
- DATA_AVAILABLE_STATUS (propagated)
- LIVELINESS_LOST_STATUS (propagated)
- LIVELINESS_CHANGED_STATUS (propagated)
- PUBLICATION_MATCHED_STATUS (propagated)
- SUBSCRIPTION_MATCHED_STATUS (propagated)

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `Listener` of a contained entity, the `Listener` on that contained entity is invoked instead of the `DomainParticipantListener`. This means that a status change on a contained entity only invokes the `DomainParticipantListener` if the contained entity itself does not handle the trigger event generated by the status change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are “Read Communication Statuses” and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a `DataReader`, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

Return Code

When the operation returns:

- *RETCODE_OK* - the DomainParticipantListener is attached
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the DomainParticipant has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.1.39 set_qos

Scope

DDS::DomainParticipant

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const DomainParticipantQos& qos);
```

Description

This operation replaces the existing set of QoSPolicy settings for a DomainParticipant.

Parameters

in const DomainParticipantQos& qos - new set of QoSPolicy settings for the DomainParticipant.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_ALREADY_DELETED* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation replaces the existing set of QoSPolicy settings for a DomainParticipant. The parameter *qos* contains 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 *RETCODE_OK*).

Return Code

When the operation returns:

- *RETCODE_OK* - the new DomainParticipantQos is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the DomainParticipant has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2 Class DomainParticipantFactory

The purpose of this class is to allow the creation and destruction of DomainParticipant objects. DomainParticipantFactory itself has no factory. It is a pre-existing singleton object that can be accessed by means of the `get_instance` operation on the DomainParticipantFactory class.

The pre-defined value `TheParticipantFactory` can also be used as an alias for the singleton factory returned by the operation `get_instance`.

The interface description of this class is as follows:

```
class DomainParticipantFactory
{
//
// implemented API operations
//
    static DomainParticipantFactory_ptr
        get_instance
            (void);
    DomainParticipant_ptr
        create_participant
            (DomainId_t domainId,
             const DomainParticipantQos& qos,
             DomainParticipantListener_ptr a_listener,
             StatusMask mask);
    ReturnCode_t
        delete_participant
            (DomainParticipant_ptr a_participant);
    DomainParticipant_ptr
        lookup_participant
            (DomainId_t domainId);
    ReturnCode_t
        set_default_participant_qos
            (const DomainParticipantQos& qos);
    ReturnCode_t
        get_default_participant_qos
            (DomainParticipantQos& qos);
    ReturnCode_t
        set_qos
            (const DomainParticipantFactoryQos& qos);
    ReturnCode_t
```

```

        get_qos
        (DomainParticipantFactoryQos& qos);
    ReturnCode_t
        delete_domain
        (Domain_ptr a_domain);
    Domain
        lookup_domain
        (const DomainId_t domainId);
};

```

The next paragraphs describe the usage of all DomainParticipantFactory operations.

3.2.2.1 create_participant

Scope

DDS::DomainParticipantFactory

Synopsis

```

#include <ccpp_dds_dcps.h>
    DomainParticipant_ptr
    create_participant
    (DomainId_t domainId,
     const DomainParticipantQos& qos,
     DomainParticipantListener_ptr a_listener,
     StatusMask mask);

```

Description

This operation creates a new DomainParticipant which will join the domain identified by domainId, with the desired DomainParticipantQos and attaches the optionally specified DomainParticipantListener to it.

Parameters

in DomainId_t domainId - the ID of the Domain to which the DomainParticipant is joined. This should be a URI to the location of the configuration file that identifies the configuration details of the Domain, or the Domain name as specified in the configuration file. The actual ID used will also be applicable for the lookup_participant and get_domain_id operations; it is not allowed to mix URI and Domain name in operations on one DomainParticipant.

in const DomainParticipantQos& qos - a DomainParticipantQos for the new DomainParticipant. When this set of QosPolicy settings is inconsistent, no DomainParticipant is created.

in DomainParticipantListener_ptr a_listener - a pointer to the DomainParticipantListener instance which will be attached to the new DomainParticipant. It is permitted to use NULL as the value of the listener: this behaves as a DomainParticipantListener whose operations perform no action.

in StatusMask mask - a bit-mask in which each bit enables the invocation of the DomainParticipantListener for a certain status.

Return Value

DomainParticipant_ptr - a pointer to the newly created DomainParticipant. In case of an error, the NULL pointer is returned.

Detailed Description

This operation creates a new DomainParticipant, with the desired DomainParticipantQos and attaches the optionally specified DomainParticipantListener to it. The DomainParticipant signifies that the calling application intends to join the Domain identified by the domainId argument.

If the specified QosPolicy settings are not consistent, the operation will fail; no DomainParticipant is created and the operation returns the NULL pointer. To delete the DomainParticipant the operation delete_participant must be used.

Identifying the Domain

The DomainParticipant will attach to the Domain that is specified by the domainId parameter. This parameter consists of a string that represents either a URI to the location of the configuration file (e.g. "file:///projects/DDS/ospl.xml"), or the Domain name as specified in the configuration file. Note that to make multiple connections to a Domain (create multiple Participants for the same Domain) within a single process, all of the Participants must use the same identification (*i.e.* all use the URI or all use the Domain name). The configuration file identified by the URI specifies all configuration details of the Domain to which it refers. See the *Deployment Guide* for further details about the contents of this configuration file.

A NULL pointer may be assigned to the DomainId: in that case the location of the configuration file is extracted from the environment variable called OSPL_URI. This variable will be initialized when you source the release.com script (on platforms to which that applies) or, on the Windows platform, when you install the OpenSplice product. Initially it will point to the default configuration file that comes with OpenSplice, but of course you are free to change this to any configuration file that you want.

It is recommended to use this `OSPL_URI` variable instead of hard-coding the URI into your application, since this gives you much more flexibility in the deployment phase of your product.

Default QoS

The constant `PARTICIPANT_QOS_DEFAULT` can be used as parameter `qos` to create a `DomainParticipant` with the default `DomainParticipantQos` as set in the `DomainParticipantFactory`. The effect of using `PARTICIPANT_QOS_DEFAULT` is the same as calling the operation `get_default_participant_qos` and using the resulting `DomainParticipantQos` to create the `DomainParticipant`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `DomainParticipantListener` operation 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 fact that the status is reset prior to calling the listener means that if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `DomainParticipantListener`:

- `INCONSISTENT_TOPIC_STATUS` *(propagated)*
- `OFFERED_DEADLINE_MISSED_STATUS` *(propagated)*
- `REQUESTED_DEADLINE_MISSED_STATUS` *(propagated)*
- `OFFERED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `SAMPLE_LOST_STATUS` *(propagated)*
- `SAMPLE_REJECTED_STATUS` *(propagated)*
- `DATA_ON_READERS_STATUS` *(propagated)*
- `DATA_AVAILABLE_STATUS` *(propagated)*
- `LIVELINESS_LOST_STATUS` *(propagated)*
- `LIVELINESS_CHANGED_STATUS` *(propagated)*
- `PUBLICATION_MATCHED_STATUS` *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS` *(propagated)*.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its

factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `Listener` of a contained entity, the `Listener` on that contained entity is invoked instead of the `DomainParticipantListener`. This means that a status change on a contained entity only invokes the `DomainParticipantListener` if the contained entity itself does not handle the trigger event generated by the status change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are “Read Communication Statuses” and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a `DataReader`, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

3.2.2.2 delete_participant

Scope

`DDS::DomainParticipantFactory`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_participant
        (DomainParticipant_ptr a_participant);
```

Description

This operation deletes a `DomainParticipant`.

Parameters

in `DomainParticipant_ptr a_participant` - a pointer to the `DomainParticipant`, which is to be deleted.

Return Value

ReturnCode_t - return codes can be `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes a `DomainParticipant`. A `DomainParticipant` cannot be deleted when it has any attached `Entity` objects. When the operation is called on a `DomainParticipant` with existing `Entity` objects, the operation returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- `RETCODE_OK` - the `DomainParticipant` is deleted
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_participant` is not a valid `DomainParticipant_ptr`
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - the `DomainParticipant` contains one or more `Entity` objects.

3.2.2.3 `get_default_participant_qos`

Scope

`DDS::DomainParticipantFactory`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_default_participant_qos
        (DomainParticipantQos& qos);
```

Description

This operation gets the default `DomainParticipantQos` of the `DomainParticipantFactory`.

Parameters

inout `DomainParticipantQos& qos` - a reference to the `DomainParticipantQos` struct (provided by the application) in which the default `DomainParticipantQos` for the `DomainParticipant` is written.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets the default `DomainParticipantQos` of the `DomainParticipantFactory` (that is the struct with the `QosPolicy` settings) which is used for newly created `DomainParticipant` objects, in case the constant `PARTICIPANT_QOS_DEFAULT` is used. The default `DomainParticipantQos` is only used when the constant is supplied as parameter `qos` to specify the `DomainParticipantQos` in the `create_participant` operation. The application must provide the `DomainParticipantQos` struct in which the `QosPolicy` settings can be stored and provide a reference to the struct. The operation writes the default `QosPolicy` settings to the struct referenced to by `qos`. Any settings in the struct are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_participant_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` setting as defined in Table 2 on page 37

Return Code

When the operation returns:

- *RETCODE_OK* - the default `DomainParticipant` `QosPolicy` settings of this `DomainParticipantFactory` have successfully been copied into the specified `DomainParticipantQos` parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.4 `get_instance`

Scope

`DDS::DomainParticipantFactory`

Synopsis

```
#include <ccpp_dds_dcps.h>
static DomainParticipantFactory_ptr
    get_instance
        (void);
```

Description

This operation returns the DomainParticipantFactory singleton.

Parameters

<none>

Return Value

DomainParticipantFactory_ptr - a pointer to the DomainParticipantFactory.

Detailed Description

This operation returns the DomainParticipantFactory singleton. The operation is idempotent, that is, it can be called multiple times without side-effects and it returns the same DomainParticipantFactory instance.

The operation is static and must be called upon its class (DomainParticipantFactory::get_instance) .

The pre-defined value TheParticipantFactory can also be used as an alias for the singleton factory returned by the operation get_instance.

3.2.2.5 get_qos**Scope**

DDS::DomainParticipantFactory

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (DomainParticipantFactoryQos& qos);
```

Description

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

Parameters

inout DomainParticipantFactoryQos& qos - a reference to the destination DomainParticipantFactoryQos struct in which the QoSPolicy settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `DomainParticipantFactory` on which this operation is used. This `DomainParticipantFactoryQos` is stored at the location pointed to by the `qos` parameter.

Return Code

When the operation returns:

- `RETCODE_OK` - the existing set of QoS policy values applied to this `DomainParticipantFactory` has successfully been copied into the specified `DomainParticipantFactoryQos` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.6 lookup_participant

Scope

`DDS::DomainParticipantFactory`

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    lookup_participant
        (DomainId_t domainId);
```

Description

This operation retrieves a previously created `DomainParticipant` belonging to the specified `domainId`.

Parameters

in const DomainId_t domainId - the ID of the Domain for which a joining `DomainParticipant` should be retrieved. This should be either a URI to the location of the configuration file that identifies the configuration details of the Domain, or the Domain name as specified in the configuration file. The actual value to be used (URI or Domain name) is dependent of the value used when creating the `DomainParticipant`, also see the

DomainParticipantFactory create_participant operation. If a DomainParticipant is created using the Domain name then the Domain name should be used to lookup the DomainParticipant, and *vice versa*: when created using a URI then the URI must be used with this operation.

Return Value

DomainParticipant_ptr - Return value is a pointer to the DomainParticipant retrieved. When no such DomainParticipant is found, the NULL pointer is returned.

Detailed Description

This operation retrieves a previously created DomainParticipant belonging to the specified domainId. If no such DomainParticipant exists, the operation will return NULL.

The domainId used to search for a specific DomainParticipant must be identical to the domainId that was used to create that specific DomainParticipant: a NULL pointer will not be resolved on this level. That means that a DomainParticipant that was created using a domainId set to NULL will not be found if you try to look it up using a hard-coded URI that has the same contents as the environment variable OSPL_URI.

The domainId used to search for a specific DomainParticipant must be identical to the domainId that was used to create that specific DomainParticipant: a NULL pointer will not be resolved on this level. This means that a DomainParticipant that was created using a domainId set to NULL will not be found if you try to look it up using a hard-coded URI that has the same contents as the environment variable OSPL_URI.

If multiple DomainParticipant entities belonging to the specified domainId exist, then the operation will return one of them. It is not specified which one.

3.2.2.7 set_default_participant_qos

Scope

DDS::DomainParticipantFactory

Synopsis

```
#include <ccpp_dps_dcps.h>
ReturnCode_t
    set_default_participant_qos
        (const DomainParticipantQos& qos);
```

Description

This operation sets the default `DomainParticipantQos` of the `DomainParticipantFactory`.

Parameters

in `const DomainParticipantQos& qos` - the `DomainParticipantQos` struct, which contains the new default `DomainParticipantQos` for the newly created `DomainParticipants`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation sets the default `DomainParticipantQos` of the `DomainParticipantFactory` (that is the struct with the `QosPolicy` settings) which is used for newly created `DomainParticipant` objects, in case the constant `PARTICIPANT_QOS_DEFAULT` is used. The default `DomainParticipantQos` is only used when the constant is supplied as parameter `qos` to specify the `DomainParticipantQos` in the `create_participant` operation. The `DomainParticipantQos` is always self consistent, because its policies do not depend on each other. This means this operation never returns the `RETCODE_INCONSISTENT_POLICY`.

The values set by this operation are returned by `get_default_participant_qos`.

Return Code

When the operation returns:

- `RETCODE_OK` - the new default `DomainParticipantQos` is set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.8 set_qos

Scope

`DDS::DomainParticipantFactory`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
```



```
set_qos
(const DomainParticipantFactoryQos& qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipantFactory`.

Parameters

in `const DomainParticipantFactoryQos& qos` - must contain the new set of `QosPolicy` settings for the `DomainParticipantFactory`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipantFactory`. The parameter `qos` must contain the struct with the `QosPolicy` settings.

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 `RETCODE_OK`).

Return Code

When the operation returns:

- `RETCODE_OK` - the new `DomainParticipantFactoryQos` is set.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.9 delete_domain

Synopsis

```
#include <ccpp-dds-dcps.h>
ReturnCode_t
delete_domain
(Domain_ptr a_domain);
```

Description

This operation deletes a Domain proxy.

Parameters

in Domain_ptr a_domain - a pointer to the Domain proxy, which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are:

RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation deletes a Domain proxy.

Return Code

When the operation returns:

- *RETCODE_OK* - the Domain proxy is deleted.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter *a_domain* is not a valid Domain proxy.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.2.2.10 lookup_domain

Synopsis

```
#include <ccpp_dds_dcps.h>
Domain_ptr
lookup_domain
    (const DomainId_t domainId);
```

Description

This operation retrieves a previously created Domain proxy belonging to the specified *domainId* or creates a new Domain proxy if no Domain proxy yet exists but the Domain itself is available.

Parameters

in const DomainId_t domainId - the ID of the Domain for which a Domain proxy should be retrieved. This should be a URI to the location of the configuration file that identifies the configuration details of the Domain.

Return Value

Domain_ptr - Return value is a pointer to the Domain proxy retrieved. When no such Domain proxy is found or could be created, the NULL pointer is returned.

Detailed Description

This operation retrieves a previously created Domain proxy belonging to the specified domainId or creates a new Domain proxy if no Domain proxy was found, but the domainId does refer to a valid Domain. If no such Domain proxy exists or could be created, the operation will return the NULL pointer.

The domainId used to search for a specific Domain proxy must not be a NULL pointer, as a NULL pointer will not be resolved on this level and in effect has no meaning.

3.2.3 Class Domain

The purpose of this class is to represent the Domain and allow certain Domain-wide operations to be performed. In essence it is a proxy to the Domain.

A Domain is a distributed concept that links all the applications that must be able to communicate with each other. It represents a communication plane: only the DDS_Publishers and the DDS_Subscribers attached to the same Domain can interact.

This class currently implements one function:

- It allows for a snapshot to be taken of all persistent data available within this Domain on local node level.

The interface description of this class is as follows:

```
/*
 * interface Domain
 */
class Domain {
    ReturnCode_t
        create_persistent_snapshot(
            const char* partition_expression,
            const char* topic_expression,
            const char* URI);
};
```

The following sections describe the usage of all Domain operations.

3.2.3.1 create_persistent_snapshot

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    create_persistent_snapshot(
```

```
const char* partition_expression,
const char* topic_expression,
const char* URI);
```

Description

This operation will create a snapshot of all persistent data matching the provided partition and topic expressions and store the snapshot at the location indicated by the URI. Only persistent data available on the local node is considered.

Parameters

in char partition_expression* - The expression of all partitions involved in the snapshot; this may contain wildcards.

in char topic_expression* - The expression of all topics involved in the snapshot; this may contain wildcards.

in char uri* - The location where to store the snapshot. Currently only directories are supported.

Return Value

ReturnCode_t - Possible return codes of the operation are:

RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation will create a snapshot of all persistent data matching the provided partition and topic expressions and store the snapshot at the location indicated by the URI. Only persistent data available on the local node is considered. This operation will fire an event to trigger the snapshot creation by the durability service and then return while the durability service fulfills the snapshot request; if no durability service is available then there is no persistent data available and the operation will return OK as a snapshot of an empty store is an empty store.

The created snapshot can then be used as the persistent store for the durability service next time it starts up by configuring the location of the snapshot as the persistent store in the configuration file. The durability service will then use the snapshot as the regular store (and can thus also alter its contents).

Return Code

When the operation returns:

- *RETCODE_OK* – The persistent snapshot is (being) created.
- *RETCODE_ERROR* - an internal error has occurred.

- `RETCODE_BAD_PARAMETER` - the parameter `partition_expression`, `topic_expression` or `uri` is NIL.
- `RETCODE_ALREADY_DELETED` - the Domain proxy has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.2.4 DomainParticipantListener interface

Since a `DomainParticipant` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `DomainParticipantListener`. This interface must be implemented by the application. A user defined class must be provided by the application which must extend from the `DomainParticipantListener` class. **All** `DomainParticipantListener` operations **must** be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.



All operations for this interface must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

The `DomainParticipantListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `DomainParticipantListener` is related to changes in communication status `StatusConditions`.

The interface description of this class is as follows:

```
class DomainParticipantListener
{
//
// inherited from TopicListener
//
// void
//   on_inconsistent_topic
//   (Topic_ptr the_topic,
//    const InconsistentTopicStatus& status) = 0;
//
// inherited from PublisherListener
//
// void
//   on_offered_deadline_missed
//   (DataWriter_ptr writer,
//    const OfferedDeadlineMissedStatus& status) = 0;

// void
//   on_offered_incompatible_qos
```

```

//      (DataWriter_ptr writer,
//      const OfferedIncompatibleQosStatus& status) = 0;

// void
//      on_liveliness_lost
//      (DataWriter_ptr writer,
//      const LivelinessLostStatus& status) = 0;

// void
//      on_publication_matched
//      (DataWriter_ptr writer,
//      const PublicationMatchedExceptionStatus& status) = 0;
//
// inherited from SubscriberListener
//
// void
//      on_data_on_readers
//      (Subscriber_ptr subs) = 0;
// void
//      on_requested_deadline_missed
//      (DataReader_ptr reader,
//      const RequestedDeadlineMissedExceptionStatus& status) = 0;

// void
//      on_requested_incompatible_qos
//      (DataReader_ptr reader,
//      const RequestedIncompatibleQosStatus& status) = 0;

// void
//      on_sample_rejected
//      (DataReader_ptr reader,
//      const SampleRejectedStatus& status) = 0;

// void
//      on_liveliness_changed
//      (DataReader_ptr reader,
//      const LivelinessChangedStatus& status) = 0;

// void
//      on_data_available
//      (DataReader_ptr reader) = 0;

// void
//      on_subscription_matched
//      (DataReader_ptr reader,
//      const SubscriptionMatchedExceptionStatus& status) = 0;

// void
//      on_sample_lost
//      (DataReader_ptr reader,

```

```

//          const SampleLostStatus& status) = 0;
//
// implemented API operations
//          <no operations>
//
};

```

The next paragraphs list all `DomainParticipantListener` operations. Since these operations are all inherited, they are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.2.4.1 `on_data_available` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_data_available
        (DataReader_ptr reader) = 0;

```

3.2.4.2 `on_data_on_readers` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `SubscriberListener` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_data_on_readers
        (Subscriber_ptr subs) = 0;

```

3.2.4.3 `on_inconsistent_topic` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `TopicListener` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_inconsistent_topic
        (Topic_ptr the_topic,
         const InconsistentTopicStatus& status) = 0;

```

3.2.4.4 on_liveliness_changed (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_liveliness_changed
        (DataReader_ptr reader,
         const LivelinessChangedStatus& status) = 0;
```

3.2.4.5 on_liveliness_lost (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_liveliness_lost
        (DataWriter_ptr writer,
         const LivelinessLostStatus& status) = 0;
```

3.2.4.6 on_offered_deadline_missed (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_offered_deadline_missed
        (DataWriter_ptr writer,
         const OfferedDeadlineMissedStatus& status) = 0;
```

3.2.4.7 on_offered_incompatible_qos (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_offered_incompatible_qos
        (DataWriter_ptr writer,
         const OfferedIncompatibleQosStatus& status) = 0;
```


3.2.4.8 on_publication_matched (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_publication_matched
        (DataWriter_ptr writer,
         const PublicationMatchedExceptionStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.2.4.9 on_requested_deadline_missed (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_requested_deadline_missed
        (DataReader_ptr reader,
         const RequestedDeadlineMissedExceptionStatus& status) = 0;
```

3.2.4.10 on_requested_incompatible_qos (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_requested_incompatible_qos
        (DataReader_ptr reader,
         const RequestedIncompatibleQosStatus& status) = 0;
```

3.2.4.11 on_sample_lost (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_sample_lost
        (DataReader_ptr reader,
         const SampleLostStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.2.4.12 on_sample_rejected (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_sample_rejected
        (DataReader_ptr reader,
         const SampleRejectedStatus& status) = 0;
```

3.2.4.13 on_subscription_matched (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_subscription_matched
        (DataReader_ptr reader,
         const SubscriptionMatchedStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.3 Topic-Definition Module

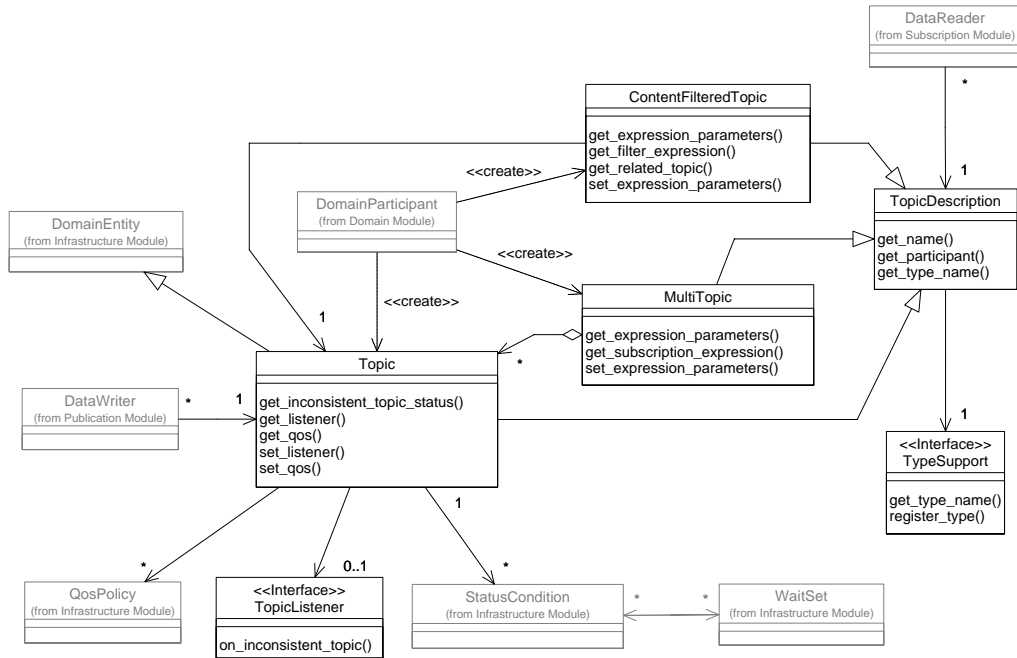


Figure 16 DCPS Topic-Definition Module’s Class Model

This module contains the following classes:

- TopicDescription (abstract)
- Topic
- ContentFilteredTopic
- MultiTopic
- TopicListener (interface)
- Topic-Definition type specific classes.

“Topic-Definition type specific classes” contains the generic class and the generated data type specific classes. For each data type, a data type specific class `<type>TypeSupport` is generated (based on IDL) by calling the pre-processor.

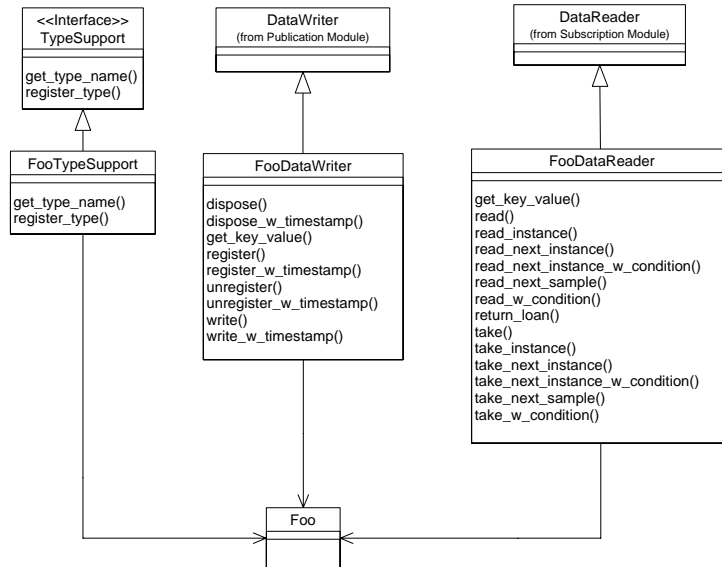


Figure 17 Data Type “Foo” Typed Classes Pre-processor Generation

For instance, for the fictional data type `Foo` (this also applies to other types) “Topic-Definition type specific classes” contains the following classes:

- `TypeSupport` (abstract)
- `FooTypeSupport`.

Topic objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A Topic is meant to fulfil that purpose: it associates a name (unique in the Domain), a data type, and `TopicQos` related to the data itself.

3.3.1 Class `TopicDescription` (abstract)

This class is an abstract class. It is the base class for `Topic`, `ContentFilteredTopic` and `MultiTopic`.

The `TopicDescription` attribute `type_name` defines an unique data type that is made available to the Data Distribution Service via the `TypeSupport`. `TopicDescription` has also a name that allows it to be retrieved locally.

The interface description of this class is as follows:

```

class TopicDescription
{
//
// implemented API operations
//
char*
    get_type_name

```

```

        (void);
char*
    get_name
        (void);
DomainParticipant_ptr
    get_participant
        (void);
};

```

The next paragraphs describe the usage of all TopicDescription operations.

3.3.1.1 `get_name`

Scope

DDS::TopicDescription

Synopsis

```

#include <ccpp_dds_dcps.h>
char*
    get_name
        (void);

```

Description

This operation returns the name used to create the TopicDescription.

Parameters

<none>

Return Value

*char** - is the name of the TopicDescription.

Detailed Description

This operation returns the name used to create the TopicDescription.

3.3.1.2 `get_participant`

Scope

DDS::TopicDescription

Synopsis

```

#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    get_participant
        (void);

```

Description

This operation returns the `DomainParticipant` associated with the `TopicDescription` or the `NULL` pointer.

Parameters

<none>

Return Value

DomainParticipant_ptr - a pointer to the `DomainParticipant` associated with the `TopicDescription` or the `NULL` pointer.

Detailed Description

This operation returns the `DomainParticipant` associated with the `TopicDescription`. Note that there is exactly one `DomainParticipant` associated with each `TopicDescription`. When the `TopicDescription` was already deleted (there is no associated `DomainParticipant` any more), the `NULL` pointer is returned.

3.3.1.3 `get_type_name`

Scope

`DDS::TopicDescription`

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_type_name
        (void);
```

Description

This operation returns the registered name of the data type associated with the `TopicDescription`.

Parameters

<none>

Return Value

*char** - the name of the data type of the `TopicDescription`.

Detailed Description

This operation returns the registered name of the data type associated with the `TopicDescription`.

3.3.2 Class Topic

Topic is the most basic description of the data to be published and subscribed.

A Topic is identified by its name, which must be unique in the whole Domain. In addition (by virtue of extending TopicDescription) it fully identifies the type of data that can be communicated when publishing or subscribing to the Topic.

Topic is the only TopicDescription that can be used for publications and therefore a specialized DataWriter is associated to the Topic.

The interface description of this class is as follows:

```
class Topic
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//   (void);
// StatusMask
//   get_status_changes
//   (void);
// ReturnCode_t
//   enable
//   (void);
//
// inherited from class TopicDescription
//
// char*
//   get_type_name
//   (void);

// char*
//   get_name
//   (void);

// DomainParticipant_ptr
//   get_participant
//   (void);
//
// implemented API operations
//
ReturnCode_t
    set_qos
        (const TopicQos& qos);
ReturnCode_t
    get_qos
        (TopicQos& qos);
ReturnCode_t
```

```

        set_listener
            (TopicListener_ptr a_listener,
             StatusMask mask);
    TopicListener_ptr
        get_listener
            (void);
    ReturnCode_t
        get_inconsistent_topic_status
            (InconsistentTopicStatus& a_status);
};

```

The next paragraphs describe the usage of all `Topic` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.3.2.1 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    enable
        (void);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.3.2.2 get_inconsistent_topic_status

Scope

`DDS::Topic`

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_inconsistent_topic_status
        (InconsistentTopicStatus& a_status);

```

Description

This operation obtains the `InconsistentTopicStatus` of the `Topic`.

Parameters

inout InconsistentTopicStatus& a_status - the contents of the `InconsistentTopicStatus` struct of the `Topic` will be copied into the location specified by *a_status*.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `InconsistentTopicStatus` of the `Topic`. The `InconsistentTopicStatus` can also be monitored using a `TopicListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `InconsistentTopicStatus` of this `Topic` has successfully been copied into the specified `a_status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `Topic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.3.2.3 `get_listener`

Scope

`DDS::Topic`

Synopsis

```
#include <ccpp_dds_dcps.h>
TopicListener_ptr
    get_listener
        (void);
```

Description

This operation allows access to a `TopicListener`.

Parameters

<none>

Return Value

TopicListener_ptr - result is a pointer to the `TopicListener` attached to the `Topic`.

Detailed Description

This operation allows access to a `TopicListener` attached to the `Topic`. When no `TopicListener` was attached to the `Topic`, the `NULL` pointer is returned.

3.3.2.4 `get_name` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_name
        (void);
```

3.3.2.5 `get_participant` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    get_participant
        (void);
```

3.3.2.6 `get_qos`

Scope

`DDS::Topic`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (TopicQos& qos);
```

Description

This operation allows access to the existing set of QoS policies for a `Topic`.

Parameters

inout `TopicQos& qos` - a reference to the destination `TopicQos` struct in which the `QosPolicy` settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation allows access to the existing set of QoS policies of a `Topic` on which this operation is used. This `TopicQos` is stored at the location pointed to by the `qos` parameter.

Return Code

When the operation returns:

- `RETCODE_OK` - the existing set of QoS policy values applied to this `Topic` has successfully been copied into the specified `TopicQos` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `Topic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.3.2.7 `get_status_changes` (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_status_changes
        (void);
```

3.3.2.8 `get_statuscondition` (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

3.3.2.9 `get_type_name` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_type_name
        (void);
```

3.3.2.10 `set_listener`

Scope

`DDS::Topic`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_listener
        (TopicListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation attaches a `TopicListener` to the `Topic`.

Parameters

in `TopicListener_ptr a_listener` - a pointer to the `TopicListener` instance, which will be attached to the `Topic`.

in `StatusMask mask` - a bit mask in which each bit enables the invocation of the `TopicListener` for a certain status.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation attaches a `TopicListener` to the `Topic`. Only one `TopicListener` can be attached to each `Topic`. If a `TopicListener` was already attached, the operation will replace it with the new one. When `a_listener` is the `NULL` pointer, it represents a listener that is treated as a NOOP¹ for all statuses activated in the bit mask.

Communication Status

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 `TopicListener` operation 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. An exception to this rule is the `NULL` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `TopicListener`:

- `INCONSISTENT_TOPIC_STATUS`.

Status bits are declared as a constant and can be used by the application in an `OR` operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

In case a communication status is not activated in the `mask` of the `TopicListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Topic` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

Return Code

When the operation returns:

- `RETCODE_OK` - the `TopicListener` is attached
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_ALREADY_DELETED` - the `Topic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

-
1. Short for **No-Operation**, an instruction that does nothing.

3.3.2.11 **set_qos**

Scope

DDS::Topic

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const TopicQos& qos);
```

Description

This operation replaces the existing set of QoSPolicy settings for a Topic.

Parameters

in const TopicQos& qos - the new set of QoSPolicy settings for the Topic.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_UNSUPPORTED, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_IMMUTABLE_POLICY or RETCODE_INCONSISTENT_POLICY.

Detailed Description

This operation replaces the existing set of QoSPolicy settings for a Topic. The parameter *qos* contains the struct with the QoSPolicy settings which is checked for self-consistency and mutability. When the application tries to change a QoSPolicy setting for an enabled Topic, which can only be set before the Topic is enabled, the operation will fail and a RETCODE_IMMUTABLE_POLICY is returned. In other words, the application must provide the currently set QoSPolicy settings in case of the immutable QoSPolicy settings. Only the mutable QoSPolicy settings can be changed. When *qos* contains conflicting QoSPolicy settings (not self-consistent), the operation will fail and a RETCODE_INCONSISTENT_POLICY is returned.

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 RETCODE_OK).

Return Code

When the operation returns:

- *RETCODE_OK* - the new TopicQos is set

- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter *qos* is not a valid *TopicQos*. It contains a *QosPolicy* setting with an invalid *Duration_t* value or an enum value that is outside its legal boundaries.
- *RETCODE_UNSUPPORTED* - one or more of the selected *QosPolicy* values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the *Topic* has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter *qos* contains an immutable *QosPolicy* setting with a different value than set during enabling of the *Topic*
- *RETCODE_INCONSISTENT_POLICY* - the parameter *qos* contains conflicting *QosPolicy* settings, e.g. a history depth that is higher than the specified resource limits.

3.3.3 Class ContentFilteredTopic

ContentFilteredTopic is a specialization of *TopicDescription* that allows for content based subscriptions.

ContentFilteredTopic describes a more sophisticated subscription that indicates the *Subscriber* does not necessarily want to see all values of each instance published under the *Topic*. Rather, it only wants to see the values whose contents satisfy certain criteria. Therefore this class must be used to request content-based subscriptions.

The selection of the content is done using the SQL based filter with parameters to adapt the filter clause.

The interface description of this class is as follows:

```
class ContentFilteredTopic
{
//
// inherited from class TopicDescription
//
// char*
//   get_type_name
//   (void);

// char*
//   get_name
//   (void);

// DomainParticipant_ptr
//   get_participant
```

```

//      (void);
//
// implemented API operations
//
char*
    get_filter_expression
        (void);

ReturnCode_t
    get_expression_parameters
        (StringSeq& expression_parameters);

ReturnCode_t
    set_expression_parameters
        (const StringSeq& expression_parameters);
Topic_ptr
    get_related_topic
        (void);
};

```

The next paragraphs describe the usage of all `ContentFilteredTopic` operations.

3.3.3.1 `get_expression_parameters`

Scope

`DDS::ContentFilteredTopic`

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_expression_parameters
        (StringSeq& expression_parameters);

```

Description

This operation obtains the expression parameters associated with the `ContentFilteredTopic`.

Parameters

inout StringSeq& expression_parameters - a reference to a sequence of strings that will be used to store the parameters used in the SQL expression.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the expression parameters associated with the `ContentFilteredTopic`. That is, the parameters specified on the last successful call to `set_expression_parameters`, or if `set_expression_parameters` was never called, the parameters specified when the `ContentFilteredTopic` was created.

The resulting reference holds a sequence of strings with the parameters used in the SQL expression (i.e., the %n tokens in the expression). The number of parameters in the result sequence will exactly match the number of %n tokens in the filter expression associated with the `ContentFilteredTopic`.

Return Code

When the operation returns:

- `RETCODE_OK` - the existing set of expression parameters applied to this `ContentFilteredTopic` has successfully been copied into the specified `expression_parameters` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `ContentFilteredTopic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.3.3.2 `get_filter_expression`

Scope

`DDS::ContentFilteredTopic`

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_filter_expression
        (void);
```

Description

This operation returns the `filter_expression` associated with the `ContentFilteredTopic`.

Parameters

<none>

Return Value

*char** - a handle to a string which holds the SQL filter expression.

Detailed Description

This operation returns the `filter_expression` associated with the `ContentFilteredTopic`. That is, the expression specified when the `ContentFilteredTopic` was created.

The filter expression result is a string that specifies the criteria to select the data samples of interest. It is similar to the `WHERE` clause of an SQL expression.

3.3.3.3 `get_name` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_name
        (void);
```

3.3.3.4 `get_participant` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    get_participant
        (void);
```

3.3.3.5 `get_related_topic`**Scope**

```
DDS::ContentFilteredTopic
```

Synopsis

```
#include <ccpp_dds_dcps.h>
Topic_ptr
    get_related_topic
        (void);
```

Description

This operation returns the `Topic` associated with the `ContentFilteredTopic`.

Parameters

<none>

Return Value

Topic_ptr - a pointer to the base topic on which the filtering will be applied.

Detailed Description

This operation returns the `Topic` associated with the `ContentFilteredTopic`. That is, the `Topic` specified when the `ContentFilteredTopic` was created. This `Topic` is the base topic on which the filtering will be applied.

3.3.3.6 `get_type_name` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_type_name
        (void);
```

3.3.3.7 `set_expression_parameters`**Scope**

`DDS::ContentFilteredTopic`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_expression_parameters
        (const StringSeq& expression_parameters);
```

Description

This operation changes the expression parameters associated with the `ContentFilteredTopic`.

Parameters

in const StringSeq& expression_parameters - a reference to a sequence of strings with the parameters used in the SQL expression (i.e., the number of %n tokens in the expression). The number of values in `expression_parameters` must be equal or greater than the highest referenced %n token in the `subscription_expression`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation changes the expression parameters associated with the `ContentFilteredTopic`. The parameter `expression_parameters` is a handle to a sequence of strings with the parameters used in the SQL expression. The number of values in `expression_parameters` must be equal or greater than the highest referenced `%n` token in the `filter_expression` (for example, if `%1` and `%8` are used as parameter in the `filter_expression`, the `expression_parameters` should at least contain `n+1 = 9` values). This is the filter expression specified when the `ContentFilteredTopic` was created.

Return Code

When the operation returns:

- `RETCODE_OK` - the new expression parameters are set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the number of parameters in `expression_parameters` does not match the number of “`%n`” tokens in the expression for this `ContentFilteredTopic` or one of the parameters is an illegal parameter
- `RETCODE_ALREADY_DELETED` - the `ContentFilteredTopic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.3.4 Class MultiTopic

`MultiTopic` is a specialization of `TopicDescription` that allows subscriptions to combine, filter and/or rearrange data coming from several `Topics`.

`MultiTopic` allows a more sophisticated subscription that can select and combine data received from multiple `Topics` into a single data type (specified by the inherited `type_name`). The data will then be filtered (selection) and possibly re-arranged (aggregation and/or projection) according to an SQL expression with parameters to adapt the filter clause.

The interface description of this class is as follows:

```
class MultiTopic
{
```

```

//
// inherited from class TopicDescription
//
// char*
//     get_type_name
//         (void);

// char*
//     get_name
//         (void);

// DomainParticipant_ptr
//     get_participant
//         (void);
//
// implemented API operations
//
char*
    get_subscription_expression
        (void);

ReturnCode_t
    get_expression_parameters
        (StringSeq& expression_parameters);

ReturnCode_t
    set_expression_parameters
        (const StringSeq& expression_parameters);
};

```

The next paragraphs describe the usage of all `MultiTopic` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

Note: `MultiTopic` operations have not been yet been implemented. Multitopic functionality is scheduled for a future release.

3.3.4.1 `get_expression_parameters`

Scope

`DDS::MultiTopic`

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_expression_parameters
        (StringSeq& expression_parameters);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation returns the expression parameters associated with the `MultiTopic`.

Parameters

inout StringSeq& expression_parameters - a reference to a sequence of strings that will be used to store the parameters used in the SQL expression.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the expression parameters associated with the `MultiTopic`. That is, the parameters specified on the last successful call to `set_expression_parameters`, or if `set_expression_parameters` was never called, the parameters specified when the `MultiTopic` was created.

The resulting reference holds a sequence of strings with the values of the parameters used in the SQL expression (i.e., the `%n` tokens in the expression). The number of parameters in the result sequence will exactly match the number of `%n` tokens in the filter expression associated with the `MultiTopic`.

Return Code

When the operation returns:

- `RETCODE_OK` - the existing set of expression parameters applied to this `MultiTopic` has successfully been copied into the specified `expression_parameters` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `MultiTopic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.3.4.2 `get_name` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_ddc_dcps.h>
char*
    get_name
```

```
(void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.3.4.3 `get_participant` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    get_participant
    (void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.3.4.4 `get_subscription_expression`

Scope

```
DDS::MultiTopic
```

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_subscription_expression
    (void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation returns the subscription expression associated with the `MultiTopic`.

Parameters

<none>

Return Value

*char** - result is a handle to a string which holds the SQL subscription expression.

Detailed Description

This operation returns the subscription expression associated with the `MultiTopic`. That is, the expression specified when the `MultiTopic` was created.

The subscription expression result is a string that specifies the criteria to select the data samples of interest. In other words, it identifies the selection and rearrangement of data from the associated `Topics`. It is an SQL expression where the `SELECT` clause provides the fields to be kept, the `FROM` part provides the names of the

Topics that are searched for those fields, and the `WHERE` clause gives the content filter. The Topics combined may have different types but they are restricted in that the type of the fields used for the `NATURAL JOIN` operation must be the same.

3.3.4.5 `get_type_name` (inherited)

This operation is inherited and therefore not described here. See the class `TopicDescription` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_type_name
        (void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.3.4.6 `set_expression_parameters`

Scope

`DDS::MultiTopic`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_expression_parameters
        (const StringSeq& expression_parameters);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation changes the expression parameters associated with the `MultiTopic`.

Parameters

in `const StringSeq& expression_parameters` - the handle to a sequence of strings with the parameters used in the SQL expression.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation changes the expression parameters associated with the `MultiTopic`. The parameter `expression_parameters` is a handle to a sequence of strings with the parameters used in the SQL expression. The number of parameters in

`expression_parameters` must exactly match the number of `%n` tokens in the subscription expression associated with the `MultiTopic`. This is the subscription expression specified when the `MultiTopic` was created.

Return Code

When the operation returns:

- `RETCODE_OK` - the new expression parameters are set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the number of parameters in `expression_parameters` does not match the number of “%n” tokens in the expression for this `MultiTopic` or one of the parameters is an illegal parameter.
- `RETCODE_ALREADY_DELETED` - the `MultiTopic` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.3.5 TopicListener interface

Since a `Topic` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `TopicListener`. This interface must be implemented by the application. A user defined class must be provided by the application which must extend from the `TopicListener` class. **All** `TopicListener` operations **must** be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.



All operations for this interface must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

The `TopicListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as an inconsistent `Topic`. The `TopicListener` is related to changes in communication status.

The interface description of this class is as follows:

```
class TopicListener
{
//
// abstract external operations
//
void
    on_inconsistent_topic
        (Topic_ptr the_topic,
         const InconsistentTopicStatus& status) = 0;
//
// implemented API operations
```

```
//      <no operations>
//
};
```

The next paragraph describes the usage of the `TopicListener` operation. This abstract operation is fully described since it must be implemented by the application.

3.3.5.1 `on_inconsistent_topic` (abstract)

Scope

```
DDS::TopicListener
```

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_inconsistent_topic
        (Topic_ptr the_topic,
         const InconsistentTopicStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `InconsistentTopicStatus` changes.

Parameters

in Topic_ptr the_topic - contain a pointer to the `Topic` on which the conflict occurred (this is an input to the application).

in const InconsistentTopicStatus& status - contain the `InconsistentTopicStatus` struct (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `InconsistentTopicStatus` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `TopicListener` is installed and enabled for the `InconsistentTopicStatus`. The `InconsistentTopicStatus` will change when another `Topic` exists with the same `topic_name` but different characteristics.

The Data Distribution Service will call the `TopicListener` operation with a parameter `the_topic`, which will contain a reference to the `Topic` on which the conflict occurred and a parameter `status`, which will contain the `InconsistentTopicStatus` struct.

3.3.6 Topic-Definition Type Specific Classes

This paragraph describes the generic `TypeSupport` class and the derived application type specific `<type>TypeSupport` classes which together implement the application `Topic` interface. For each application type, used as `Topic` data type, the pre-processor generates a `<type>DataReader` class from an IDL type description. The `FooTypeSupport` class that would be generated by the pre-processor for a fictional type `Foo` describes the `<type>TypeSupport` classes.

3.3.6.1 Class `TypeSupport` (abstract)

The `Topic`, `MultiTopic` or `ContentFilteredTopic` is bound to a data type described by the type name argument. Prior to creating a `Topic`, `MultiTopic` or `ContentFilteredTopic`, the data type must have been registered with the Data Distribution Service. This is done using the data type specific `register_type` operation on a derived class of the `TypeSupport` interface. A derived class is generated for each data type used by the application, by calling the pre-processor.

The interface description of this class is as follows:

```
class TypeSupport
{
//
// abstract operations
//
// ReturnCode_t
//   register_type
//       (Domainparticipant_ptr domain,
//        const char* type_name);
// char*
//   get_type_name
//       (void);
//
// implemented API operations
//   <no operations>
//
};
```

The next paragraph list the `TypeSupport` operation. This abstract operation is listed but not fully described since it is not implemented in this class. The full description of this operation is given in the `FooTypeSupport` class (for the data type example `Foo`), which contains the data type specific implementation of this operation.

3.3.6.2 `get_type_name` (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>TypeSupport` class. For further explanation see the description for the fictional data type `Foo` derived `FooTypeSupport` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_type_name
        (void);
```

3.3.6.3 `register_type` (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>TypeSupport` class. For further explanation see the description for the fictional data type `Foo` derived `FooTypeSupport` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    register_type
        (Domainparticipant_ptr domain,
         const char* type_name);
```

3.3.6.4 Class `FooTypeSupport`

The pre-processor generates from IDL type descriptions the application `<type>TypeSupport` classes. For each application data type that is used as `Topic` data type, a typed class `<type>TypeSupport` is derived from the `TypeSupport` class. In this paragraph, the class `FooTypeSupport` describes the operations of these derived `<type>TypeSupport` classes as an example for the fictional application type `Foo` (defined in the module `SPACE`).

For instance, for an application, the definitions are located in the `Space.idl` file. The pre-processor will generate a `ccpp_Space.h` include file.



General note: The name `ccpp_Space.h` is derived from the IDL file `Space.idl`, that defines `SPACE::Foo`, for all relevant `SPACE::FooDataWriter` operations.

The `Topic`, `MultiTopic` or `ContentFilteredTopic` is bound to a data type described by the `type_name` argument. Prior to creating a `Topic`, `MultiTopic` or `ContentFilteredTopic`, the data type must have been registered with the Data Distribution Service. This is done using the data type specific `register_type` operation on the `<type>TypeSupport` class for each data type. A derived class is generated for each data type used by the application, by calling the pre-processor.

The interface description of this class is as follows:

```

class FooTypeSupport
{
//
// implemented API operations
//
    ReturnCode_t
        register_type
            (DomainParticipant_ptr domain,
             const char* type_name);
    char*
        get_type_name
            (void);
};

```

The next paragraph describes the usage of the `FooTypeSupport` operation.

3.3.6.5 `get_type_name`

Scope

`SPACE::FooTypeSupport`

Synopsis

```

#include <ccpp_Space.h>
char*
    get_type_name
        (void);

```

Description

This operation returns the default name of the data type associated with the `FooTypeSupport`.

Parameters

<none>

Return Value

*char** - the name of the data type of the `FooTypeSupport`.

Detailed Description

This operation returns the default name of the data type associated with the `FooTypeSupport`. The default name is derived from the type name as specified in the IDL definition. It is composed of the scope names and the type name, each separated by “: :”, in order of lower scope level to deeper scope level followed by the type name.

3.3.6.6 register_type

Scope

SPACE::FooTypeSupport

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    register_type
        (DomainParticipant_ptr domain,
         const char* type_name);
```

Description

This operation registers a new data type name to a DomainParticipant.

Parameters

in DomainParticipant_ptr domain - a pointer to a DomainParticipant object to which the new data type is registered.

in const char type_name* - a local alias of the new data type to be registered.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation registers a new data type name to a DomainParticipant. This operation informs the Data Distribution Service, in order to allow it to manage the new registered data type. This operation also informs the Data Distribution Service about the key definition, which allows the Data Distribution Service to distinguish different instances of the same data type.

Precondition

A *type_name* cannot be registered with two different <type>TypeSupport classes (this means of a different data type) with the same DomainParticipant. When the operation is called on the same DomainParticipant with the same *type_name* for a different <type>TypeSupport class, the operation returns RETCODE_PRECONDITION_NOT_MET. However, it is possible to register the same <type>TypeSupport classes with the same DomainParticipant and the same or different *type_name* multiple times. All registrations return RETCODE_OK, but any subsequent registrations with the same *type_name* are ignored.

Return Code

When the operation returns:

- *RETCODE_OK* - the FooTypeSupport class is registered with the new data type name to the DomainParticipant or the FooTypeSupport class was already registered
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the domain parameter is a NULL pointer or the parameter type_name has zero length
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation
- *RETCODE_PRECONDITION_NOT_MET* - this type_name is already registered with this DomainParticipant for a different <type>TypeSupport class.

3.4 Publication Module

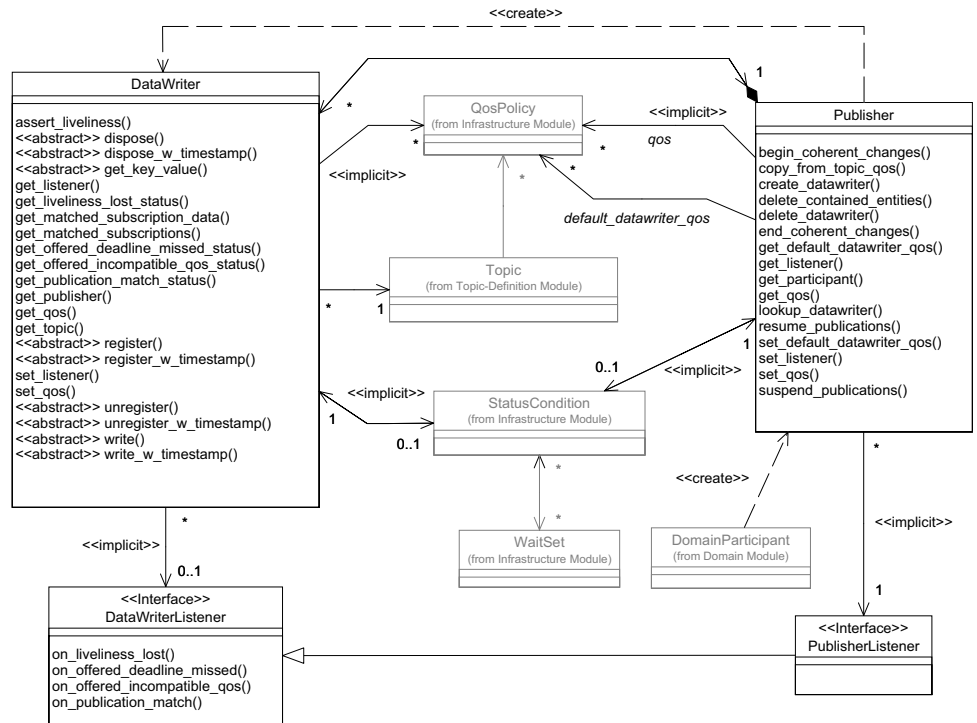


Figure 18 DCPS Publication Module's Class Model

This module contains the following classes:

- Publisher
- Publication type specific classes
- PublisherListener (interface)
- DataWriterListener (interface).

The paragraph “Publication type specific classes” contains the generic class and the generated data type specific classes. For each data type, a data type specific class `<type>DataWriter` is generated (based on IDL) by calling the pre-processor.

For instance, for the fictional data type `Foo` (this also applies to other types) “Publication type specific classes” contains the following classes:

- `DataWriter` (abstract)
- `FooDataWriter`.

A `Publisher` is an object responsible for data distribution. It may publish data of different data types. A `DataWriter` acts as a typed accessor to a `Publisher`. The `DataWriter` is the object the application must use to communicate the existence and value of data-objects of a given data type to a `Publisher`. When data-object values have been communicated to the `Publisher` through the appropriate `DataWriter`, it is the `Publisher`’s responsibility to perform the distribution. The `Publisher` will do this according to its own `PublisherQos`, and the `DataWriterQos` attached to the corresponding `DataWriter`. A publication is defined by the association of a `DataWriter` to a `Publisher`. This association expresses the intent of the application to publish the data described by the `DataWriter` in the context provided by the `Publisher`.

3.4.1 Class Publisher

The `Publisher` acts on behalf of one or more `DataWriter` objects that belong to it. When it is informed of a change to the data associated with one of its `DataWriter` objects, it decides when it is appropriate to actually process the sample-update message. In making this decision, it considers the `PublisherQos` and the `DataWriterQos`.

The interface description of this class is as follows:

```
class Publisher
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//   (void);
// StatusMask
```



```

//      get_status_changes
//      (void);
//      ReturnCode_t
//      enable
//      (void);
//
// implemented API operations
//
    DataWriter_ptr
        create_datawriter
            (Topic_ptr a_topic,
             const DataWriterQos& qos,
             DataWriterListener_ptr a_listener,
             StatusMask mask);

    ReturnCode_t
        delete_datawriter
            (DataWriter_ptr a_datawriter);

    DataWriter_ptr
        lookup_datawriter
            (const char* topic_name);
    ReturnCode_t
        delete_contained_entities
            (void);

    ReturnCode_t
        set_qos
            (const PublisherQos& qos);
    ReturnCode_t
        get_qos
            (PublisherQos& qos);
    ReturnCode_t
        set_listener
            (PublisherListener_ptr a_listener,
             StatusMask mask);
    PublisherListener_ptr
        get_listener
            (void);
    ReturnCode_t
        suspend_publications
            (void);

    ReturnCode_t
        resume_publications
            (void);

    ReturnCode_t
        begin_coherent_changes
            (void);

```

```

        ReturnCode_t
            end_coherent_changes
                (void);

        ReturnCode_t
            wait_for_acknowledgments
                (const Duration_t& max_wait);

        DomainParticipant_ptr
            get_participant
                (void);

        ReturnCode_t
            set_default_datawriter_qos
                (const DataWriterQos& qos);

        ReturnCode_t
            get_default_datawriter_qos
                (DataWriterQos& qos);

        ReturnCode_t
            copy_from_topic_qos
                (DataWriterQos& a_datawriter_qos,
                 const TopicQos& a_topic_qos);
    };

```

The next paragraphs describe the usage of all `Publisher` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.4.1.1 `begin_coherent_changes`

Scope

`DDS::Publisher`

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    begin_coherent_changes
        (void);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation requests that the application will begin a ‘coherent set’ of modifications using `DataWriter` objects attached to this `Publisher`. The ‘coherent set’ will be completed by a matching call to `end_coherent_changes`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are:
`RETCODE_UNSUPPORTED`.

Detailed Description

This operation requests that the application will begin a ‘coherent set’ of modifications using `DataWriter` objects attached to this `Publisher`. The ‘coherent set’ will be completed by a matching call to `end_coherent_changes`.

A ‘coherent set’ is a set of modifications that must be propagated in such a way that they are interpreted at the receivers’ side as a consistent set of modifications; that is, the receiver will only be able to access the data after all the modifications in the set are available at the receiver end.

A connectivity change may occur in the middle of a set of coherent changes; for example, the set of partitions used by the `Publisher` or one of its connected `Subscribers` may change, a late-joining `DataReader` may appear on the network, or a communication failure may occur. In the event that such a change prevents an entity from receiving the entire set of coherent changes, that entity must behave as if it had received none of the set.

These calls can be nested. In that case, the coherent set terminates only with the last call to `end_coherent_changes`.

The support for ‘coherent changes’ enables a publishing application to change the value of several data-instances that could belong to the same or different topics and have those changes be seen ‘atomically’ by the readers. This is useful in cases where the values are inter-related (for example, if there are two data-instances representing the ‘altitude’ and ‘velocity vector’ of the same aircraft and both are changed, it may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

Return Code

When the operation returns:

- `RETCODE_UNSUPPORTED` - the operation is not yet implemented. It is scheduled for a future release.

3.4.1.2 copy_from_topic_qos

Scope

DDS::Publisher

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    copy_from_topic_qos
        (DataWriterQos& a_datawriter_qos,
         const TopicQos& a_topic_qos);
```

Description

This operation will copy policies in `a_topic_qos` to the corresponding policies in `a_datawriter_qos`.

Parameters

inout DataWriterQos& a_datawriter_qos - the destination DataWriterQos struct to which the QosPolicy settings should be copied.

in const TopicQos& a_topic_qos - the source TopicQos struct, which should be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation will copy the QosPolicy settings in `a_topic_qos` to the corresponding QosPolicy settings in `a_datawriter_qos` (replacing the values in `a_datawriter_qos`, if present). This will only apply to the common QosPolicy settings in each <Entity>Qos.

This is a “convenience” operation, useful in combination with the operations `get_default_datawriter_qos` and `Topic::get_qos`. The operation `copy_from_topic_qos` can be used to merge the DataWriter default QosPolicy settings with the corresponding ones on the TopicQos. The resulting DataWriterQos can then be used to create a new DataWriter, or set its DataWriterQos.

This operation does not check the resulting `a_datawriter_qos` for consistency. This is because the “merged” `a_datawriter_qos` may not be the final one, as the application can still modify some `QosPolicy` settings prior to applying the `DataWriterQos` to the `DataWriter`.

Return Code

When the operation returns:

- `RETCODE_OK` - the `QosPolicy` settings are copied from the `Topic` to the `DataWriter`
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `Publisher` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the `Data Distribution Service` ran out of resources to complete this operation.

3.4.1.3 `create_datawriter`

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
DataWriter_ptr
    create_datawriter
        (Topic_ptr a_topic,
         const DataWriterQos& qos,
         DataWriterListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation creates a `DataWriter` with the desired `DataWriterQos`, for the desired `Topic` and attaches the optionally specified `DataWriterListener` to it.

Parameters

in `Topic_ptr a_topic` - a pointer to the `topic` for which the `DataWriter` is created.

in `const DataWriterQos& qos` - the `DataWriterQos` for the new `DataWriter`. In case these settings are not self consistent, no `DataWriter` is created.

in DataWriterListener_ptr a_listener - a pointer to the DataWriterListener instance which will be attached to the new DataWriter. It is permitted to use NULL as the value of the listener: this behaves as a DataWriterListener whose operations perform no action.

Return Value

DataWriter_ptr - Return value is a pointer to the newly created DataWriter. In case of an error, the NULL pointer is returned.

Detailed Description

This operation creates a DataWriter with the desired DataWriterQos, for the desired Topic and attaches the optionally specified DataWriterListener to it. The returned DataWriter is attached (and belongs) to the Publisher on which this operation is being called. To delete the DataWriter the operation `delete_datawriter` or `delete_contained_entities` must be used. If no write rights are defined for the specific topic then the creation of the DataWriter will fail.

Application Data Type

The DataWriter returned by this operation is an object of a derived class, specific to the data type associated with the Topic. For each application-defined data type `<type>` there is a class `<type>DataWriter` generated by calling the pre-processor. This data type specific class extends DataWriter and contains the operations to write data of data type `<type>`.

QosPolicy

The possible application pattern to construct the DataWriterQos for the DataWriter is to:

- Retrieve the QosPolicy settings on the associated Topic by means of the `get_qos` operation on the Topic
- Retrieve the default DataWriterQos by means of the `get_default_datawriter_qos` operation on the Publisher
- Combine those two lists of QosPolicy settings and selectively modify QosPolicy settings as desired
- Use the resulting DataWriterQos to construct the DataWriter.

In case the specified QosPolicy settings are not consistent, no DataWriter is created and the NULL pointer is returned.

Default QoS

The constant `DATAWRITER_QOS_DEFAULT` can be used as parameter `qos` to create a `DataWriter` with the default `DataWriterQos` as set in the `Publisher`. The effect of using `DATAWRITER_QOS_DEFAULT` is the same as calling the operation `get_default_datawriter_qos` and using the resulting `DataWriterQos` to create the `DataWriter`.

The special `DATAWRITER_QOS_USE_TOPIC_QOS` can be used to create a `DataWriter` with a combination of the default `DataWriterQos` and the `TopicQos`. The effect of using `DATAWRITER_QOS_USE_TOPIC_QOS` is the same as calling the operation `get_default_datawriter_qos` and retrieving the `TopicQos` (by means of the operation `Topic::get_qos`) and then combining these two `QosPolicy` settings using the operation `copy_from_topic_qos`, whereby any common policy that is set on the `TopicQos` “overrides” the corresponding policy on the default `DataWriterQos`. The resulting `DataWriterQos` is then applied to create the `DataWriter`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `DataWriterListener` operation 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 fact that the status is reset prior to calling the listener means that if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `DataWriterListener`:

- `OFFERED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `LIVELINESS_LOST_STATUS`
- `PUBLICATION_MATCHED_STATUS`.

Status bits are declared as a constant and can be used by the application in an `OR` operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

In case a communication status is not activated in the mask of the `DataWriterListener`, the `PublisherListener` of the containing `Publisher` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `PublisherListener` of the containing `Publisher` and a `DataWriter` specific behaviour when needed. In case the communication status is not activated in the mask of the `PublisherListener` as well, the communication status will be propagated to the `DomainParticipantListener` of the containing `DomainParticipant`. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

3.4.1.4 delete_contained_entities**Scope**

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_contained_entities
        (void);
```

Description

This operation deletes all the `DataWriter` objects that were created by means of one of the `create_datawriter` operations on the `Publisher`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes all the `DataWriter` objects that were created by means of one of the `create_datawriter` operations on the `Publisher`. In other words, it deletes all contained `DataWriter` objects.



Note: The operation will return `PRECONDITION_NOT_MET` if the any of the contained entities is in a state where it cannot be deleted. In such cases, the operation does not roll-back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

- `RETCODE_OK` - the contained Entity objects are deleted and the application may delete the Publisher
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the Publisher has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - one or more of the contained entities are in a state where they cannot be deleted.

3.4.1.5 delete_datawriter

Scope

DDS::Publisher

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_datawriter
        (DataWriter_ptr a_datawriter);
```

Description

This operation deletes a `DataWriter` that belongs to the Publisher.

Parameters

in `DataWriter_ptr a_datawriter` - a pointer to the `DataWriter`, which is to be deleted.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes a `DataWriter` that belongs to the Publisher. When the operation is called on a different Publisher, as used when the `DataWriter` was created, the operation has no effect and returns `RETCODE_PRECONDITION_NOT_MET`. The deletion of the `DataWriter` will automatically unregister all instances. Depending on the settings of `WriterDataLifecycleQosPolicy`, the deletion of the `DataWriter` may also dispose of all instances.

Return Code

When the operation returns:

- `RETCODE_OK` - the `DataWriter` is deleted
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_datawriter` is not a valid `DataWriter_ptr`
- `RETCODE_ALREADY_DELETED` - the Publisher has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_PRECONDITION_NOT_MET` - the operation is called on a different Publisher, as used when the `DataWriter` was created.

3.4.1.6 **enable (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    enable
        (void);
```

3.4.1.7 **end_coherent_changes**

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    end_coherent_changes
        (void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation terminates the ‘coherent set’ initiated by the matching call to `begin_coherent_changes`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are:
`RETCODE_UNSUPPORTED`.

Detailed Description

This operation terminates the ‘coherent set’ initiated by the matching call to `Publisher_begin_coherent_changes`. If there is no matching call to `Publisher_begin_coherent_changes`, the operation will return the error `PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- `RETCODE_UNSUPPORTED` - the operation is not yet implemented. It is scheduled for a future release.

3.4.1.8 `get_default_datawriter_qos`

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_default_datawriter_qos
        (DataWriterQos& qos);
```

Description

This operation gets the default `DataWriterQos` of the `Publisher`.

Parameters

inout DataWriterQos& qos - a reference to the `DataWriterQos` struct (provided by the application) in which the default `DataWriterQos` for the `DataWriter` is written.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation gets the default `DataWriterQos` of the Publisher (that is the struct with the `QosPolicy` settings) which is used for newly created `DataWriter` objects, in case the constant `DATAWRITER_QOS_DEFAULT` is used. The default `DataWriterQos` is only used when the constant is supplied as parameter `qos` to specify the `DataWriterQos` in the `create_datawriter` operation. The application must provide the `DataWriterQos` struct in which the `QosPolicy` settings can be stored and pass the `qos` reference to the operation. The operation writes the default `DataWriterQos` to the struct referenced to by `qos`. Any settings in the struct are overwritten.

The values retrieved by this operation match the set of values specified on the last successful call to `set_default_datawriter_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` setting as defined in Table 2 on page 37.

Return Code

When the operation returns:

- *RETCODE_OK* - the default `DataWriter QosPolicy` settings of this Publisher have successfully been copied into the specified `DataWriterQos` parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the Publisher has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.1.9 get_listener

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
PublisherListener_ptr
    get_listener
        (void);
```

Description

This operation allows access to a `PublisherListener`.

Parameters

<none>

Return Value

PublisherListener_ptr - result is a pointer to the `PublisherListener` attached to the `Publisher`.

Detailed Description

This operation allows access to a `PublisherListener` attached to the `Publisher`. When no `PublisherListener` was attached to the `Publisher`, the `NULL` pointer is returned.

3.4.1.10 `get_participant`

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    get_participant
        (void);
```

Description

This operation returns the `DomainParticipant` associated with the `Publisher` or the `NULL` pointer.

Parameters

<none>

Return Value

DomainParticipant_ptr - a pointer to the `DomainParticipant` associated with the `Publisher` or the `NULL` pointer.

Detailed Description

This operation returns the `DomainParticipant` associated with the `Publisher`. Note that there is exactly one `DomainParticipant` associated with each `Publisher`. When the `Publisher` was already deleted (there is no associated `DomainParticipant` any more), the `NULL` pointer is returned.

3.4.1.11 `get_qos`

Scope

DDS::Publisher

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (PublisherQos& qos);
```

Description

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

Parameters

inout PublisherQos& qos - a reference to the destination PublisherQos struct in which the QosPolicy settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation allows access to the existing set of QoS policies of a Publisher on which this operation is used. This PublisherQos is stored at the location pointed to by the qos parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this Publisher has successfully been copied into the specified PublisherQos parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the Publisher has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.1.12 `get_status_changes` (inherited)

This operation is inherited and therefore not described here. See the class Entity for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_status_changes
        (void);
```

3.4.1.13 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

3.4.1.14 **lookup_datawriter**

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
DataWriter_ptr
    lookup_datawriter
        (const char* topic_name);
```

Description

This operation returns a previously created `DataWriter` belonging to the `Publisher` which is attached to a `Topic` with the matching `topic_name`.

Parameters

in `const char* topic_name` - the name of the `Topic`, which is attached to the `DataWriter` to look for.

Return Value

`DataWriter_ptr` - Return value is a pointer to the `DataWriter` found. When no such `DataWriter` is found, the `NULL` pointer is returned.

Detailed Description

This operation returns a previously created `DataWriter` belonging to the `Publisher` which is attached to a `Topic` with the matching `topic_name`. When multiple `DataWriter` objects (which satisfy the same condition) exist, this operation will return one of them. It is not specified which one.

3.4.1.15 `resume_publications`

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    resume_publications
        (void);
```

Description

This operation resumes a previously suspended publication.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED` or `RETCODE_PRECONDITION_NOT_MET`

Detailed Description

If the `Publisher` is suspended, this operation will resume the publication of all `DataWriter` objects contained by this `Publisher`. All data held in the history buffer of the `DataWriter`'s is actively published to the consumers. When the operation returns all `DataWriter`'s have resumed the publication of suspended updates.

Return Code

When the operation returns:

- *RETCODE_OK* - the `Publisher` has been suspended
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the `Publisher` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the Publisher is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the Publisher is not suspended

3.4.1.16 **set_default_datawriter_qos**

Scope

DDS::Publisher

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_default_datawriter_qos
        (const DataWriterQos& qos);
```

Description

This operation sets the default *DataWriterQos* of the Publisher.

Parameters

in const DataWriterQos& qos - the *DataWriterQos* struct, which contains the new default *DataWriterQos* for the newly created *DataWriters*.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_UNSUPPORTED*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES* or *RETCODE_INCONSISTENT_POLICY*.

Detailed Description

This operation sets the default *DataWriterQos* of the Publisher (that is the struct with the *QosPolicy* settings) which is used for newly created *DataWriter* objects, in case the constant *DATAWRITER_QOS_DEFAULT* is used. The default *DataWriterQos* is only used when the constant is supplied as parameter *qos* to specify the *DataWriterQos* in the *create_datawriter* operation. The *set_default_datawriter_qos* operation checks if the *DataWriterQos* is self consistent. If it is not, the operation has no effect and returns *RETCODE_INCONSISTENT_POLICY*.

The values set by this operation are returned by *get_default_datawriter_qos*.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default *DataWriterQos* is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter *qos* is not a valid *DataWriterQos*. It contains a *QosPolicy* setting with an invalid *Duration_t* value or an enum value that is outside its legal boundaries.
- *RETCODE_ALREADY_DELETED* - the *Publisher* has already been deleted
- *RETCODE_INCONSISTENT_POLICY* - the parameter *qos* contains conflicting *QosPolicy* settings, e.g. a history depth that is higher than the specified resource limits.

3.4.1.17 set_listener

Scope

DDS::Publisher

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_listener
        (PublisherListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation attaches a *PublisherListener* to the *Publisher*.

Parameters

in PublisherListener_ptr a_listener - a pointer to the *PublisherListener* instance, which will be attached to the *Publisher*.

in StatusMask mask - a bit mask in which each bit enables the invocation of the *PublisherListener* for a certain status.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_ALREADY_DELETED* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation attaches a `PublisherListener` to the `Publisher`. Only one `PublisherListener` can be attached to each `Publisher`. If a `PublisherListener` was already attached, the operation will replace it with the new one. When a `_listener` is the `NULL` pointer, it represents a listener that is treated as a NOOP¹ for all statuses activated in the bit mask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `PublisherListener` operation 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>_status` from inside the listener it will see the status already reset. An exception to this rule is the `NULL` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `PublisherListener`:

- `OFFERED_DEADLINE_MISSED_STATUS` *(propagated)*
- `OFFERED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `LIVELINESS_LOST_STATUS` *(propagated)*
- `PUBLICATION_MATCHED_STATUS` *(propagated).*

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataWriterListener` of a contained `DataWriter`, the `DataWriterListener` on that contained `DataWriter` is invoked instead of the `PublisherListener`. This means, that a status change on a contained `DataWriter` only invokes the `PublisherListener` if the contained `DataWriter` itself does not handle the trigger event generated by the status change.

1. Short for **No-Operation**, an instruction that does nothing.

In case a status is not activated in the mask of the `PublisherListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Publisher` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

Return Code

When the operation returns:

- `RETCODE_OK` - the `PublisherListener` is attached
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_ALREADY_DELETED` - the `Publisher` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.4.1.18 `set_qos`

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const PublisherQos& qos);
```

Description

This operation replaces the existing set of `QosPolicy` settings for a `Publisher`.

Parameters

in `const PublisherQos& qos` - the new set of `QosPolicy` settings for the `Publisher`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, or `RETCODE_IMMUTABLE_POLICY` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation replaces the existing set of `QoSPolicy` settings for a `Publisher`. The parameter `qos` contains the `QoSPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QoSPolicy` setting for an enabled `Publisher`, which can only be set before the `Publisher` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the currently set `QoSPolicy` settings in case of the immutable `QoSPolicy` settings. Only the mutable `QoSPolicy` settings can be changed. When `qos` contains conflicting `QoSPolicy` settings (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

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 `RETCODE_OK`). If one or more of the partitions in the `QoS` structure have insufficient access rights configured then the `set_qos` function will fail with a `DDS_RETCODE_PRECONDITION_NOT_MET` error code.

Return Code

When the operation returns:

- `RETCODE_OK` - the new `PublisherQoS` is set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `qos` is not a valid `PublisherQoS`. It contains a `QoSPolicy` setting with an enum value that is outside its legal boundaries.
- `RETCODE_UNSUPPORTED` - one or more of the selected `QoSPolicy` values are currently not supported by `OpenSplice`.
- `RETCODE_ALREADY_DELETED` - the `Publisher` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_IMMUTABLE_POLICY` - the parameter `qos` contains an immutable `QoSPolicy` setting with a different value than set during enabling of the `Publisher`.
- `RETCODE_PRECONDITION_NOT_MET` - returned when insufficient access rights exist for the partition(s) listed in the `QoS` structure.

3.4.1.19 `suspend_publications`

Scope

`DDS::Publisher`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    suspend_publications
        (void);
```

Description

This operation will suspend the dissemination of the publications by all contained *DataWriter* objects.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES* or *RETCODE_NOT_ENABLED*.

Detailed Description

This operation suspends the publication of all *DataWriter* objects contained by this *Publisher*. The data written, disposed or unregistered by a *DataWriter* is stored in the history buffer of the *DataWriter* and therefore, depending on its QoS settings, the following operations may block (see the operation descriptions for more information):

- *DDS::DataWriter.dispose*
- *DDS::DataWriter.dispose_w_timestamp*
- *DDS::DataWriter.write*
- *DDS::DataWriter.write_w_timestamp*
- *DDS::DataWriter.writedispose*
- *DDS::DataWriter.writedispose_w_timestamp*
- *DDS::DataWriter.unregister_instance*
- *DDS::DataWriter.unregister_instance_w_timestamp*

Subsequent calls to this operation have no effect. When the *Publisher* is deleted before *resume_publications* is called, all suspended updates are discarded.

Return Code

When the operation returns:

- *RETCODE_OK* - the *Publisher* has been suspended
- *RETCODE_ERROR* - an internal error has occurred

- *RETCODE_ALREADY_DELETED* - the Publisher has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the Publisher is not enabled.

3.4.1.20 wait_for_acknowledgments

Scope

DDS::Publisher

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    wait_for_acknowledgments
        (const Duration_t& max_wait);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation blocks the calling thread until either all data written by the contained DataWriters is acknowledged by the matched DataReaders, or until the duration specified by *max_wait* parameter elapses, whichever happens first.

Parameters

in const Duration_t& max_wait - the maximum duration to block for the *wait_for_acknowledgments*, after which the application thread is unblocked. The special constant *DURATION_INFINITE* can be used when the maximum waiting time does not need to be bounded.

Return Value

ReturnCode_t - Possible return codes of the operation are:
RETCODE_UNSUPPORTED.

Detailed Description

This operation is intended to be used only if one or more of the contained DataWriters has its *ReliabilityQosPolicyKind* set to *RELIABLE_RELIABILITY_QOS*. Otherwise the operation will return immediately with *RETCODE_OK*.

It blocks the calling thread until either all data written by the contained reliable DataWriters is acknowledged by all matched DataReader entities that have their *ReliabilityQosPolicyKind* set to *RELIABLE_RELIABILITY_QOS*, or else the duration specified by the *max_wait* parameter elapses, whichever happens first. A return value of *RETCODE_OK* indicates that all the samples written have been

acknowledged by all reliable matched data readers; a return value of `RETCODE_TIMEOUT` indicates that `max_wait` elapsed before all the data was acknowledged.

Return Code

When the operation returns:

- `RETCODE_UNSUPPORTED` - the operation is not yet implemented. It is scheduled for a future release.

3.4.2 Publication Type Specific Classes

This paragraph describes the generic `DataWriter` class and the derived application type specific `<type>DataWriter` classes which together implement the application publication interface. For each application type, used as `Topic` data type, the pre-processor generates a `<type>DataWriter` class from an IDL type description. The `FooDataWriter` class that would be generated by the pre-processor for a fictional type `Foo` describes the `<type>DataWriter` classes.

3.4.2.1 Class `DataWriter` (abstract)

`DataWriter` allows the application to set the value of the sample to be published under a given `Topic`.

A `DataWriter` is attached to exactly one `Publisher` which acts as a factory for it.

A `DataWriter` is bound to exactly one `Topic` and therefore to exactly one data type. The `Topic` must exist prior to the `DataWriter`'s creation.

`DataWriter` is an abstract class. It must be specialized for each particular application data type. For a fictional application data type `Foo` (defined in the module `SPACE`) the specialized class would be `SPACE::FooDataWriter`.

The interface description of this class is as follows:

```
class DataWriter
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//   (void);
// StatusMask
//   get_status_changes
//   (void);
// ReturnCode_t
//   enable
//   (void);
//
// abstract operations (implemented in the data type specific
```



```

// DataWriter)
//
// InstanceHandle_t
//   register_instance
//     (const <data>& instance_data);
//
// InstanceHandle_t
//   register_instance_w_timestamp
//     (const <data>& instance_data,
//      const Time_t& source_timestamp);
//
// ReturnCode_t
//   unregister_instance
//     (const <data>& instance_data,
//      InstanceHandle_t handle);
//
// ReturnCode_t
//   unregister_instance_w_timestamp
//     (const <data>& instance_data,
//      InstanceHandle_t handle,
//      const Time_t& source_timestamp);
//
// ReturnCode_t
//   write
//     (const <data>& instance_data,
//      InstanceHandle_t handle);
//
// ReturnCode_t
//   write_w_timestamp
//     (const <data>& instance_data,
//      InstanceHandle_t handle,
//      const Time_t& source_timestamp);
//
// ReturnCode_t
//   dispose
//     (const <data>& instance_data,
//      InstanceHandle_t instance_handle);
//
// ReturnCode_t
//   dispose_w_timestamp
//     (const <data>& instance_data,
//      InstanceHandle_t instance_handle,
//      const Time_t& source_timestamp);
//
// ReturnCode_t
//   writedispose
//     (const <data>& instance_data,
//      InstanceHandle_t instance_handle);
//
// ReturnCode_t

```

```

//      writedispose_w_timestamp
//      (const <data>& instance_data,
//      InstanceHandle_t instance_handle,
//      const Time_t& source_timestamp);
//
// ReturnCode_t
//      get_key_value
//      (<data>& key_holder,
//      InstanceHandle_t handle);
//
// InstanceHandle_t
//      lookup_instance
//      (const <data>& instance_data);
//
// implemented API operations
//
ReturnCode_t
    set_qos
    (const DataWriterQos& qos);

ReturnCode_t
    get_qos
    (DataWriterQos& qos);

ReturnCode_t
    set_listener
    (DataWriterListener_ptr a_listener,
    StatusMask mask);

DataWriterListener_ptr
    get_listener
    (void);

Topic_ptr
    get_topic
    (void);

Publisher_ptr
    get_publisher
    (void);

ReturnCode_t
    wait_for_acknowledgments
    (const Duration_t& max_wait);

ReturnCode_t
    get_liveliness_lost_status
    (LivelinessLostStatus& status);

ReturnCode_t

```

```

        get_offered_deadline_missed_status
            (OfferedDeadlineMissedStatus& status);

    ReturnCode_t
        get_offered_incompatible_qos_status
            (OfferedIncompatibleQosStatus& status);

    ReturnCode_t
        get_publication_matched_status
            (PublicationMatchedStatus& status);

    ReturnCode_t
        assert_liveliness
            (void);

    ReturnCode_t
        get_matched_subscriptions
            (InstanceHandleSeq& subscription_handles);

    ReturnCode_t
        get_matched_subscription_data
            (SubscriptionBuiltinTopicData& subscription_data,
             InstanceHandle_t subscription_handle);
};

```

The next paragraphs describe the usage of all `DataWriter` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited. The abstract operations are listed but not fully described because they are not implemented in this specific class. The full description of these operations is located in the subclasses, which contain the data type specific implementation of these operations.

3.4.2.2 `assert_liveliness`

Scope

DDS::DataWriter

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
    assert_liveliness
        (void);

```

Description

This operation asserts the liveliness for the `DataWriter`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

Detailed Description

This operation will manually assert the liveness for the `DataWriter`. This way, the Data Distribution Service is informed that the corresponding `DataWriter` is still alive. This operation is used in combination with the `LivelinessQosPolicy` set to `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS` or `MANUAL_BY_TOPIC_LIVELINESS_QOS`. See Section 3.1.3.10, *LivelinessQosPolicy*, on page 57, for more information on `LivelinessQosPolicy`.

Writing data via the `write` operation of a `DataWriter` will assert the liveness on the `DataWriter` itself and its containing `DomainParticipant`. Therefore, `assert_liveliness` is only needed when **not** writing regularly.

The liveness should be asserted by the application, depending on the `LivelinessQosPolicy`. Asserting the liveness for this `DataWriter` can also be achieved by asserting the liveness to the `DomainParticipant`.

Return Code

When the operation returns:

- `RETCODE_OK` - the liveness of this `DataWriter` has successfully been asserted.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataWriter` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `DataWriter` is not enabled.

3.4.2.3 dispose (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
    ReturnCode_t
    dispose
        (const <data>& instance_data,
         InstanceHandle_t instance_handle);
```

3.4.2.4 dispose_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
    ReturnCode_t
    dispose_w_timestamp
        (const <data>& instance_data,
         InstanceHandle_t instance_handle,
         const Time_t& source_timestamp);
```

3.4.2.5 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
    ReturnCode_t
    enable
        (void);
```

3.4.2.6 get_key_value (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
    ReturnCode_t
    get_key_value
        (<data>& key_holder,
```

```
InstanceHandle_t handle);
```

3.4.2.7 get_listener

Scope

```
DDS::DataWriter
```

Synopsis

```
#include <ccpp_dds_dcps.h>
DataWriterListener_ptr
    get_listener
        (void);
```

Description

This operation allows access to a `DataWriterListener`.

Parameters

<none>

Return Value

DataWriterListener_ptr - result is a pointer to the `DataWriterListener` attached to the `DataWriter`.

Detailed Description

This operation allows access to a `DataWriterListener` attached to the `DataWriter`. When no `DataWriterListener` was attached to the `DataWriter`, the `NULL` pointer is returned.

3.4.2.8 get_liveliness_lost_status

Scope

```
DDS::DataWriter
```

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_liveliness_lost_status
        (LivelinessLostStatus& status);
```

Description

This operation obtains the `LivelinessLostStatus` struct of the `DataWriter`.

Parameters

inout LivelinessLostStatus& status - the contents of the LivelinessLostStatus struct of the DataWriter will be copied into the location specified by status.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation obtains the LivelinessLostStatus struct of the DataWriter. This struct contains the information whether the liveliness (that the DataWriter has committed through its LivelinessQosPolicy) was respected.

This means, that the status represents whether the DataWriter failed to actively signal its liveliness within the offered liveliness period. If the liveliness is lost, the DataReader objects will consider the DataWriter as no longer “alive”.

The LivelinessLostStatus can also be monitored using a DataWriterListener or by using the associated StatusCondition.

Return Code

When the operation returns:

- *RETCODE_OK* - the current LivelinessLostStatus of this DataWriter has successfully been copied into the specified status parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the DataWriter has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.9 `get_matched_subscription_data`

Scope

DDS::DataWriter

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_matched_subscription_data
        (SubscriptionBuiltinTopicData& subscription_data,
         InstanceHandle_t subscription_handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.4.2.10 `get_matched_subscriptions`

Scope

DDS::DataWriter

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_matched_subscriptions
        (InstanceHandleSeq& subscription_handles);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.4.2.11 `get_offered_deadline_missed_status`

Scope

DDS::DataWriter

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_offered_deadline_missed_status
        (OfferedDeadlineMissedStatus& status);
```

Description

This operation obtains the `OfferedDeadlineMissedStatus` struct of the `DataWriter`.

Parameters

inout OfferedDeadlineMissedStatus& status - the contents of the `OfferedDeadlineMissedStatus` struct of the `DataWriter` will be copied into the location specified by `status`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `OfferedDeadlineMissedStatus` struct of the `DataWriter`. This struct contains the information whether the deadline (that the `DataWriter` has committed through its `DeadlineQosPolicy`) was respected for each instance.

The `OfferedDeadlineMissedStatus` can also be monitored using a `DataWriterListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `LivelinessLostStatus` of this `DataWriter` has successfully been copied into the specified `status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataWriter` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.12 `get_offered_incompatible_qos_status`

Scope

`DDS::DataWriter`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_offered_incompatible_qos_status
        (OfferedIncompatibleQosStatus& status);
```

Description

This operation obtains the `OfferedIncompatibleQosStatus` struct of the `DataWriter`.

Parameters

inout `OfferedIncompatibleQosStatus& status` - the contents of the `OfferedIncompatibleQosStatus` struct of the `DataWriter` will be copied into the location specified by `status`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `OfferedIncompatibleQosStatus` struct of the `DataWriter`. This struct contains the information whether a `QosPolicy` setting was incompatible with the requested `QosPolicy` setting.

This means, that the status represents whether a `DataReader` object has been discovered by the `DataWriter` with the same `Topic` and a requested `DataReaderQos` that was incompatible with the one offered by the `DataWriter`.

The `OfferedIncompatibleQosStatus` can also be monitored using a `DataWriterListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `OfferedIncompatibleQosStatus` of this `DataWriter` has successfully been copied into the specified status parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataWriter` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.13 `get_publication_matched_status`

Scope

`DDS::DataWriter`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_publication_matched_status
        (PublicationMatchedStatus& status);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.4.2.14 `get_publisher`

Scope

`DDS::DataWriter`

Synopsis

```
#include <ccpp_dds_dcps.h>
Publisher_ptr
    get_publisher
        (void);
```

Description

This operation returns the `Publisher` to which the `DataWriter` belongs.

Parameters

<none>

Return Value

Publisher_ptr - Return value is a pointer to the Publisher to which the *DataWriter* belongs.

Detailed Description

This operation returns the Publisher to which the *DataWriter* belongs, thus the Publisher that has created the *DataWriter*. If the *DataWriter* is already deleted, the *NULL* pointer is returned.

3.4.2.15 get_qos**Scope**

DDS::DataWriter

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (DataWriterQos& qos);
```

Description

This operation allows access to the existing list of *QosPolicy* settings for a *DataWriter*.

Parameters

inout DataWriterQos& qos - a reference to the destination *DataWriterQos* struct in which the *QosPolicy* settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_ALREADY_DELETED* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation allows access to the existing list of *QosPolicy* settings of a *DataWriter* on which this operation is used. This *DataWriterQos* is stored at the location pointed to by the *qos* parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this *DataWriter* has successfully been copied into the specified *DataWriterQos* parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the *DataWriter* has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.16 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class *Entity* for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_status_changes
        (void);
```

3.4.2.17 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the class *Entity* for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

3.4.2.18 get_topic**Scope**

```
DDS::DataWriter
```

Synopsis

```
#include <ccpp_dds_dcps.h>
Topic_ptr
    get_topic
        (void);
```

Description

This operation returns the *Topic* which is associated with the *DataWriter*.

Parameters

<none>

Return Value

Topic_ptr - Return value is a pointer to the *Topic* which is associated with the *DataWriter*.

Detailed Description

This operation returns the *Topic* which is associated with the *DataWriter*, thus the *Topic* with which the *DataWriter* is created. If the *DataWriter* is already deleted, the *NULL* pointer is returned.

3.4.2.19 lookup_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the <type>*DataWriter* class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type *Foo* derived *FooDataWriter* class.

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceHandle_t
lookup_instance
    (const <data>& instance_data);
```

3.4.2.20 register_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the <type>*DataWriter* class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type *Foo* derived *FooDataWriter* class.

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceHandle_t
register_instance
    (const <data>& instance_data);
```

3.4.2.21 register_instance_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceHandle_t
register_instance_w_timestamp
(const <data>& instance_data,
 const Time_t& source_timestamp);
```

3.4.2.22 set_listener

Scope

`DDS::DataWriter`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
set_listener
(DataWriterListener_ptr a_listener,
 StatusMask mask);
```

Description

This operation attaches a `DataWriterListener` to the `DataWriter`.

Parameters

in `DataWriterListener_ptr a_listener` - a pointer to the `DataWriterListener` instance, which will be attached to the `DataWriter`.

in `StatusMask mask` - a bit mask in which each bit enables the invocation of the `DataWriterListener` for a certain status.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation attaches a `DataWriterListener` to the `DataWriter`. Only one `DataWriterListener` can be attached to each `DataWriter`. If a `DataWriterListener` was already attached, the operation will replace it with the new one. When a `_listener` is the `NULL` pointer, it represents a listener that is treated as a NOOP¹ for all statuses activated in the bit mask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `DataWriterListener` operation 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>_status` from inside the listener it will see the status already reset. An exception to this rule is the `NULL` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `DataWriterListener`:

- `OFFERED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `LIVELINESS_LOST_STATUS`
- `PUBLICATION_MATCHED_STATUS`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The DDS will therefore attempt to propagate these statuses to its factory. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

In case a communication status is not activated in the mask of the `DataWriterListener`, the `PublisherListener` of the containing `Publisher` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `PublisherListener` of the containing `Publisher` and a `DataWriter` specific behaviour when needed. In case the communication status is not activated in the mask of the `PublisherListener` as well, the communication status will be propagated to the

1. Short for **No-Operation**, an instruction that does nothing.

DomainParticipantListener of the containing DomainParticipant. In case the DomainParticipantListener is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

Return Code

When the operation returns:

- *RETCODE_OK* - the DataWriterListener is attached
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the DataWriter has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.4.2.23 set_qos

Scope

DDS::DataWriter

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const DataWriterQos& qos);
```

Description

This operation replaces the existing set of QosPolicy settings for a DataWriter.

Parameters

in const DataWriterQos& qos - new set of QosPolicy settings for the DataWriter.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_UNSUPPORTED*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES*, *RETCODE_IMMUTABLE_POLICY* or *RETCODE_INCONSISTENT_POLICY*.

Detailed Description

This operation replaces the existing set of QosPolicy settings for a DataWriter. The parameter *qos* contains the struct with the QosPolicy settings which is checked for self-consistency and mutability. When the application tries to change a QosPolicy setting for an enabled DataWriter, which can only be set before the

`DataWriter` is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the presently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` settings (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

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 `RETCODE_OK`).

Return Code

When the operation returns:

- `RETCODE_OK` - the new default `DataWriterQos` is set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `qos` is not a valid `DataWriterQos`. It contains a `QosPolicy` setting with an invalid `Duration_t` value or an enum value that is outside its legal boundaries.
- `RETCODE_UNSUPPORTED` - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.
- `RETCODE_ALREADY_DELETED` - the `DataWriter` has already been deleted
- `RETCODE_IMMUTABLE_POLICY` - the parameter `qos` contains an immutable `QosPolicy` setting with a different value than set during enabling of the `DataWriter`.
- `RETCODE_INCONSISTENT_POLICY` - the parameter `qos` contains an inconsistent `QosPolicy` settings, e.g. a history depth that is higher than the specified resource limits.

3.4.2.24 unregister_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
unregister_instance
    (const <data>& instance_data,
     InstanceHandle_t handle);
```

3.4.2.25 unregister_instance_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    unregister_instance_w_timestamp
        (const <data>& instance_data,
         InstanceHandle_t handle,
         const Time_t& source_timestamp);
```

3.4.2.26 wait_for_acknowledgments

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    wait_for_acknowledgments
        (const Duration_t& max_wait);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

Description

This operation blocks the calling thread until either all data written by the `DataWriter` is acknowledged by the matched `DataReaders`, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Parameters

in `const Duration_t& max_wait` - the maximum duration to block for the `wait_for_acknowledgments`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

`ReturnCode_t` - Possible return codes of the operation are: `RETCODE_UNSUPPORTED`.

Detailed Description

This operation is intended to be used only if the `DataWriter` has its `ReliabilityQosPolicyKind` set to `RELIABLE_RELIABILITY_QOS`. Otherwise the operation will return immediately with `RETCODE_OK`.

It blocks the calling thread until either all data written by the `DataWriter` is acknowledged by all matched `DataReader` entities that have their `ReliabilityQosPolicyKind` set to `RELIABLE_RELIABILITY_QOS`, or else the duration specified by the `max_wait` parameter elapses, whichever happens first. A return value of `RETCODE_OK` indicates that all the samples written have been acknowledged by all reliable matched data readers; a return value of `RETCODE_TIMEOUT` indicates that `max_wait` elapsed before all the data was acknowledged.

Return Code

When the operation returns:

- `RETCODE_UNSUPPORTED` - the operation is not yet implemented. It is scheduled for a future release.

3.4.2.27 **write (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
write
    (const <data>& instance_data,
     InstanceHandle_t handle);
```

3.4.2.28 **write_w_timestamp (abstract)**

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
write_w_timestamp
    (const <data>& instance_data,
     InstanceHandle_t handle,
     const Time_t& source_timestamp);
```

3.4.2.29 writedispose (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
writedispose
(const <data>& instance_data,
 InstanceHandle_t handle);
```

3.4.2.30 writedispose_w_timestamp (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataWriter` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataWriter` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
writedispose_w_timestamp
(const <data>& instance_data,
 InstanceHandle_t handle,
 const Time_t& source_timestamp);
```

3.4.2.31 Class FooDataWriter

The pre-processor generates from IDL type descriptions the application `<type>DataWriter` classes. For each application data type that is used as `Topic` data type, a typed class `<type>DataWriter` is derived from the `DataWriter` class. In this paragraph, the class `FooDataWriter` in the namespace `SPACE` describes the operations of these derived `<type>DataWriter` classes as an example for the fictional application type `Foo` (defined in the module `SPACE`).

For instance, for an application, the definitions are located in the `Space.idl` file. The pre-processor will generate a `ccpp_Space.h` include file.



General note: The name `ccpp_Space.h` is derived from the IDL file `Space.idl`, that defines `SPACE::Foo`, for all relevant `SPACE::FooDataWriter` operations.

A `FooDataWriter` is attached to exactly one `Publisher` which acts as a factory for it. The `FooDataWriter` is bound to exactly one `Topic` that has been registered to use a data type `Foo`. The `Topic` must exist prior to the `FooDataWriter` creation.

The interface description of this class is as follows:

```
class FooDataWriter
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//       (void);
// StatusMask
//   get_status_changes
//       (void);
// ReturnCode_t
//   enable
//       (void);
//
// inherited from class DataWriter
//
// ReturnCode_t
//   set_qos
//       (const DataWriterQos& qos);

// ReturnCode_t
//   get_qos
//       (DataWriterQos& qos);

// ReturnCode_t
//   set_listener
//       (DataWriterListener_ptr a_listener,
//       StatusMask mask);

// DataWriterListener_ptr
//   get_listener
//       (void);

// Topic_ptr
//   get_topic
//       (void);

// Publisher_ptr
//   get_publisher
//       (void);

// ReturnCode_t
```

```

//      wait_for_acknowledgments
//      (const Duration_t& max_wait);

// ReturnCode_t
//      get_liveliness_lost_status
//      (LivelinessLostStatus& status);

// ReturnCode_t
//      get_offered_deadline_missed_status
//      (OfferedDeadlineMissedStatus& status);

// ReturnCode_t
//      get_offered_incompatible_qos_status
//      (OfferedIncompatibleQosStatus& status);

// ReturnCode_t
//      get_publication_matched_status
//      (PublicationMatchedStatus& status);

// ReturnCode_t
//      assert_liveliness
//      (void);

// ReturnCode_t
//      get_matched_subscriptions
//      (InstanceHandleSeq& subscription_handles);

// ReturnCode_t
//      get_matched_subscription_data
//      (SubscriptionBuiltinTopicData& subscription_data,
//      InstanceHandle_t subscription_handle);
//
// implemented API operations
//
InstanceHandle_t
    register_instance
        (const Foo& instance_data);
InstanceHandle_t
    register_instance_w_timestamp
        (const Foo& instance_data,
         const Time_t& time_stamp);
ReturnCode_t
    unregister_instance
        (const Foo& instance_data,
         InstanceHandle_t handle);
ReturnCode_t
    unregister_instance_w_timestamp
        (const Foo& instance_data,
         InstanceHandle_t handle,
         const Time_t& time_stamp);

```

```

        ReturnCode_t
        write
            (const Foo& instance_data,
             InstanceHandle_t handle);
        ReturnCode_t
        write_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t handle,
             const Time_t& time_stamp);
        ReturnCode_t
        dispose
            (const Foo& instance_data,
             InstanceHandle_t instance_handle);
        ReturnCode_t
        dispose_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t instance_handle,
             const Time_t& time_stamp);
        ReturnCode_t
        writedispose
            (const Foo& instance_data,
             InstanceHandle_t instance_handle);
        ReturnCode_t
        writedispose_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t instance_handle,
             const Time_t& time_stamp);
        ReturnCode_t
        get_key_value
            (Foo& key_holder,
             InstanceHandle_t handle);
        InstanceHandle_t
        lookup_instance
            (const Foo& instance_data);
    };

```

The next paragraphs describe the usage of all `FooDataWriter` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.4.2.32 `assert_liveliness` (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```

#include <ccpp_Space.h>
ReturnCode_t

```

```
assert_liveliness
(void);
```

3.4.2.33 dispose

Scope

```
SPACE::FooDataWriter
```

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
dispose
(const Foo& instance_data,
 InstanceHandle_t instance_handle);
```

Description

This operation requests the Data Distribution Service to mark the instance for deletion.

Parameters

in const Foo& instance_data - the actual instance to be disposed of.

in InstanceHandle_t instance_handle - the handle to the instance to be disposed of.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_TIMEOUT.

Detailed Description

This operation requests the Data Distribution Service to mark the instance for deletion. Copies of the instance and its corresponding samples, which are stored in every connected `DataReader` and, dependent on the `QoSPolicy` settings, also in the Transient and Persistent stores, will be marked for deletion by setting their `InstanceStateKind` to `NOT_ALIVE_DISPOSED_INSTANCE_ STATE`.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQoSPolicy`.

As a side effect, this operation asserts liveliness on the `DataWriter` itself and on the containing `DomainParticipant`.

Effects on DataReaders

Actual deletion of the instance administration in a connected `DataReader` will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it.
 - A `DataWriter` can register an instance explicitly by using one of the special operations `register_instance` or `register_instance_w_timestamp`.
 - A `DataWriter` can register an instance implicitly by using the special constant `HANDLE_NIL` in any of the other `DataWriter` operations.
 - A `DataWriter` can unregister an instance explicitly by using one of the special operations `unregister_instance` or `unregister_instance_w_timestamp`.
 - A `DataWriter` will unregister all its contained instances implicitly when it is deleted.
 - When a `DataReader` detects a loss of liveness in one of its connected `DataWriters`, it will consider all instances registered by that `DataWriter` as being implicitly unregistered.
- **and** the application must have consumed all samples belonging to the instance, either implicitly or explicitly.
 - An application can consume samples explicitly by invoking the `take` operation, or one of its variants, on its `DataReaders`.
 - The `DataReader` can consume disposed samples implicitly when the `autopurge_disposed_samples_delay` of the `ReaderDataLifecycleQosPolicy` has expired.

The `DataReader` may also remove instances that haven't been disposed first: this happens when the `autopurge_nowriter_samples_delay` of the `ReaderDataLifecycleQosPolicy` has expired after the instance is considered unregistered by all connected `DataWriters` (i.e. when it has a `InstanceStateKind` of `NOT_ALIVE_NO_WRITERS`). See also Section 3.1.3.15, *ReaderDataLifecycleQosPolicy*, on page 65.

Effects on Transient/Persistent Stores

Actual deletion of the instance administration in the connected Transient and Persistent stores will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it. (See above.)

- **and** the period of time specified by the `service_cleanup_delay` attribute in the `DurabilityServiceQosPolicy` on the Topic must have elapsed after the instance is considered unregistered by all connected `DataWriters`.

See also Section 3.1.3.4, *DurabilityServiceQosPolicy*, on page 48.

Instance Handle

The `HANDLE_NIL` handle value can be used for the parameter `instance_handle`. This indicates the identity of the instance is automatically deduced from the `instance_data` (by means of the key).

If `instance_handle` is any value other than `HANDLE_NIL`, it must correspond to the value that was returned by either the `register_instance` operation or the `register_instance_w_timestamp` operation, when the instance (identified by its key) was registered. If there is no correspondence, the result of the operation is unspecified.

The sample that is passed as `instance_data` is only used to check for consistency between its key values and the supplied `instance_handle`: the sample itself will not actually be delivered to the connected `DataReaders`. Use the `writedispose` operation if the sample itself should be delivered together with the dispose request.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the dispose operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the dispose operation may block (waiting for space to become available). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits, the `SPACE_FooDataWriter_dispose` operation will fail and returns `RETCODE_TIMEOUT`.

Sample Validation

OpenSplice DDS offers the possibility to check the sample that is passed as `instance_data` for validity. Because validity checking might reduce the overall performance, it is by default disabled. This has been done by enclosing the validity checking with conditional compiler directives like this:

```
#ifdef OSPL_BOUNDS_CHECK
    // check a specific bound.
#endif
```



By defining a macro called `OSPL_OSPL_BOUNDS_CHECK`, the validity checking will be included. On most compilers this macro can be defined by passing an additional command line parameter called `-DOSPL_BOUNDS_CHECK`.

Since the `dispose` operation merely uses the sample to check for consistency between its key values and the supplied `instance_handle`, only these keyfields will be validated against the restrictions imposed by the IDL to C++ language mapping, where:

- an enum may not exceed the value of its highest label
- a string (bounded or unbounded) may not be NULL. (Use "" for an empty string instead)
- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated when validity checking is enabled, the operation will fail and return a `RETCODE_BAD_PARAMETER`. More specific information about the context of this error will be written to the error log. When validity checking is disabled, any of these violations may result in undefined behaviour.

Return Code

When the operation returns:

- `RETCODE_OK` - the Data Distribution Service is informed that the instance data must be disposed of
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_BAD_PARAMETER` - `instance_handle` is not a valid handle or `instance_data` is not a valid sample.
- `RETCODE_ALREADY_DELETED` - the `FooDataWriter` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataWriter` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - the `instance_handle` has not been registered with this `FooDataWriter`.
- `RETCODE_TIMEOUT` - the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`. This caused blocking of the `dispose` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.34 `dispose_w_timestamp`

Scope

SPACE::FooDataWriter

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
        dispose_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t instance_handle,
             const Time_t& source_timestamp);
```

Description

This operation requests the Data Distribution Service to mark the instance for deletion and provides a value for the `source_timestamp` explicitly.

Parameters

in const Foo& instance_data - the actual instance to be disposed of.

in InstanceHandle_t instance_handle - the handle to the instance to be disposed of.

in Time_t source_timestamp - `source_timestamp` is the timestamp which is provided for the DataReader.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation performs the same functions as `dispose` except that the application provides the value for the `source_timestamp` that is made available to connected DataReader objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service is informed that the instance data must be disposed of
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - `instance_handle` is not a valid handle or `instance_data` is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the *FooDataWriter* is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the *instance_handle* has not been registered with this *FooDataWriter*.
- *RETCODE_TIMEOUT* - the current action overflowed the available resources as specified by the combination of the *ReliabilityQosPolicy*, *HistoryQosPolicy* and *ResourceLimitsQosPolicy*. This caused blocking of the *dispose_w_timestamp* operation, which could not be resolved before *max_blocking_time* of the *ReliabilityQosPolicy* elapsed.

3.4.2.35 enable (inherited)

This operation is inherited and therefore not described here. See the class *Entity* for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    enable
        (void);
```

3.4.2.36 get_key_value

Scope

SPACE::FooDataWriter

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_key_value
        (Foo& key_holder,
         InstanceHandle_t handle);
```

Description

This operation retrieves the key value of a specific instance.

Parameters

inout Foo& key_holder - a reference to the sample in which the key values are stored.

in InstanceHandle_t handle - the handle to the instance from which to get the key value.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation retrieves the key value of the instance referenced to by `instance_handle`. When the operation is called with a `HANDLE_NIL` handle value as an `instance_handle`, the operation will return `RETCODE_BAD_PARAMETER`. The operation will only fill the fields that form the key inside the `key_holder` instance. This means, the non-key fields are not applicable and may contain garbage.

The operation must only be called on registered instances. Otherwise the operation returns the error `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- *RETCODE_OK* - the `key_holder` instance contains the key values of the instance;
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - `handle` is not a valid handle
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - this instance is not registered.

3.4.2.37 `get_listener` (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
DataWriterListener_ptr
    get_listener
        (void);
```

3.4.2.38 `get_liveliness_lost_status` (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_liveliness_lost_status
        (LivelinessLostStatus& status);
```

3.4.2.39 get_matched_subscription_data (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_matched_subscription_data
        (SubscriptionBuiltinTopicData& subscription_data,
         InstanceHandle_t subscription_handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.4.2.40 get_matched_subscriptions (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_matched_subscriptions
        (InstanceHandleSeq& subscription_handles);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.4.2.41 get_offered_deadline_missed_status (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_offered_deadline_missed_status
        (OfferedDeadlineMissedStatus& status);
```

3.4.2.42 get_offered_incompatible_qos_status (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_offered_incompatible_qos_status
        (OfferedIncompatibleQosStatus& status);
```

3.4.2.43 get_publication_matched_status (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_publication_matched_status
        (PublicationMatchedStatus& status);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.4.2.44 get_publisher (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
Publisher_ptr
    get_publisher
        (void);
```

3.4.2.45 get_qos (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_qos
        (DataWriterQos& qos);
```

3.4.2.46 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
StatusMask
```



```
get_status_changes
(void);
```

3.4.2.47 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
StatusCondition_ptr
get_statuscondition
(void);
```

3.4.2.48 **get_topic (inherited)**

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
Topic_ptr
get_topic
(void);
```

3.4.2.49 **lookup_instance**

Scope

```
SPACE::FooDataWriter
```

Synopsis

```
#include <ccpp_Space.h>
InstanceHandle_t
lookup_instance
(const Foo& instance_data);
```

Description

This operation returns the value of the instance handle which corresponds to the `instance_data`.

Parameters

in `const Foo& instance_data` - a reference to the instance for which the corresponding instance handle needs to be looked up.

Return Value

InstanceHandle_t - Result value is the instance handle which corresponds to the *instance_data*.

Detailed Description

This operation returns the value of the instance handle which corresponds to the *instance_data*. The *instance_data* parameter is only used for the purpose of examining the fields that define the key. The instance handle can be used in any write, dispose or unregister operations (or their time stamped variants) that operate on a specific instance. Note that *DataWriter* instance handles are local, and are not interchangeable with *DataReader* instance handles nor with instance handles of an other *DataWriter*.

This operation does not register the instance in question. If the instance has not been previously registered, if the *DataWriter* is already deleted or if for any other reason the Service is unable to provide an instance handle, the Service will return the special value *HANDLE_NIL*.

Sample Validation

OpenSplice DDS offers the possibility to check the sample that is passed as *instance_data* for validity. Because validity checking might reduce the overall performance, it is by default disabled. This has been done by enclosing the validity checking with conditional compiler directives like this:

```
#ifdef OSPL_BOUNDS_CHECK
    // check a specific bound.
#endif
```

By defining a macro called *OSPL_BOUNDS_CHECK*, the validity checking will be included. On most compilers this macro can be defined by passing an additional command line parameter called *-DOSPL_BOUNDS_CHECK*.

Since the *lookup_instance* operation merely uses the sample to determine its identity based on the uniqueness of its key values, only the keyfields will be validated against the restrictions imposed by the IDL to C++ language mapping:

- an enum may not exceed the value of its highest label
- a string (bounded or unbounded) may not be *NULL*. (Use "" for an empty string instead)
- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated when validity checking is enabled, the operation will fail and return a *HANDLE_NIL*. More specific information about the context of this error will be written to the error log. When validity checking is disabled, any of these violations may result in undefined behaviour.

3.4.2.50 register_instance

Scope

SPACE::FooDataWriter

Synopsis

```
#include <ccpp_Space.h>
InstanceHandle_t
register_instance
(const Foo& instance_data);
```

Description

This operation informs the Data Distribution Service that the application will be modifying a particular instance.

Parameters

in const Foo& instance_data - the instance, which the application writes to or disposes of.

Return Value

InstanceHandle_t - Result value is the handle to the instance, which may be used for writing and disposing of. In case of an error, a HANDLE_NIL handle value is returned.

Detailed Description

This operation informs the Data Distribution Service that the application will be modifying a particular instance. This operation may be invoked prior to calling any operation that modifies the instance, such as `write`, `write_w_timestamp`, `unregister_instance`, `unregister_instance_w_timestamp`, `dispose`, `dispose_w_timestamp`, `writediscard` and `writediscard_w_timestamp`. When the application does register the instance before modifying, the Data Distribution Service will handle the instance more efficiently. It takes as a parameter (*instance_data*) an instance (to get the key value) and returns a handle that can be used in successive DataWriter operations. In case of an error, a HANDLE_NIL handle value is returned.

The explicit use of this operation is optional as the application can directly call the `write`, `write_w_timestamp`, `unregister_instance`, `unregister_instance_w_timestamp`, `dispose`, `dispose_w_timestamp`, `writediscard` and `writediscard_w_timestamp` operations and specify a HANDLE_NIL handle value to indicate that the sample should be examined to identify the instance.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `register_instance` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. In case the synchronous attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `register_instance` operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the `register_instance` operation will fail and returns `HANDLE_NIL`.

Sample Validation

OpenSplice DDS offers the possibility to check the sample that is passed as `instance_data` for validity. Because validity checking might reduce the overall performance, it is by default disabled. This has been done by enclosing the validity checking with conditional compiler directives like this:

```
#ifdef OSPL_BOUNDS_CHECK
    // check a specific bound.
#endif
```

i

By defining a macro called `OSPL_OSPL_BOUNDS_CHECK`, the validity checking will be included. On most compilers this macro can be defined by passing an additional command line parameter called `-DOSPL_BOUNDS_CHECK`.

Since the `register_instance` operation merely uses the sample to determine its identity based on the uniqueness of its `key` values, only the `keyfields` will be validated against the restrictions imposed by the IDL to C++ language mapping:

- an enum may not exceed the value of its highest label
- a string (bounded or unbounded) may not be `NULL`. (Use `"` for an empty string instead)
- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated when validity checking is enabled, the operation will fail and return a `HANDLE_NIL`. More specific information about the context of this error will be written to the error log. When validity checking is disabled, any of these violations may result in undefined behaviour.

Multiple Calls

If this operation is called for an already registered instance, it just returns the already allocated instance handle. This may be used to look up and retrieve the handle allocated to a given instance.

3.4.2.51 `register_instance_w_timestamp`

Scope

`SPACE::FooDataWriter`

Synopsis

```
#include <ccpp_Space.h>
        InstanceHandle_t
        register_instance_w_timestamp
        (const Foo& instance_data,
         const Time_t& source_timestamp);
```

Description

This operation will inform the Data Distribution Service that the application will be modifying a particular instance and provides a value for the `source_timestamp` explicitly.

Parameters

in Foo instance_data - the instance, which the application will write to or dispose of.

in const Time_t& source_timestamp - the timestamp used.

Return Value

InstanceHandle_t - Result value is the handle to the Instance, which must be used for writing and disposing. In case of an error, a `HANDLE_NIL` handle value is returned.

Detailed Description

This operation performs the same functions as `register_instance` except that the application provides the value for the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Multiple Calls

If this operation is called for an already registered instance, it just returns the already allocated instance handle. The `source_timestamp` is ignored in that case.

3.4.2.52 set_listener (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    set_listener
        (DataWriterListener_ptr a_listener,
         StatusMask mask);
```

3.4.2.53 set_qos (inherited)

This operation is inherited and therefore not described here. See the class `DataWriter` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    set_qos
        (const DataWriterQos& qos);
```

3.4.2.54 unregister_instance**Scope**

```
SPACE::FooDataWriter
```

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    unregister_instance
        (const Foo& instance_data,
         InstanceHandle_t handle);
```

Description

This operation informs the Data Distribution Service that the application will **not** be modifying a particular instance any more.

Parameters

in const Foo& instance_data - the instance to which the application was writing or disposing.

in *InstanceHandle_t* *handle* - the handle to the instance, which has been used for writing and disposing.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation informs the Data Distribution Service that the application will **not** be modifying a particular instance any more. Therefore, this operation reverses the action of `register_instance` or `register_instance_w_timestamp`. It should only be called on an instance that is currently registered. This operation should be called just once per instance, regardless of how many times `register_instance` was called for that instance. This operation also indicates that the Data Distribution Service can locally remove all information regarding that instance. The application should not attempt to use the `handle`, previously allocated to that instance, after calling this operation.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Effects

If, after unregistering, the application wants to modify (write or dispose) the instance, it has to register the instance again, or it has to use the special handle value `HANDLE_NIL`.

This operation does not indicate that the instance should be deleted (that is the purpose of dispose). This operation just indicates that the `DataWriter` no longer has “anything to say” about the instance. If there is no other `DataWriter` that has registered the instance as well, then the `InstanceStateKind` in all connected `DataReaders` will be changed to `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`, provided this `InstanceStateKind` was not already set to `NOT_ALIVE_DISPOSED_INSTANCE_STATE`. In the last case the `InstanceStateKind` will not be effected by the `unregister_instance` call, see also Figure 21:, *State Chart of the instance_state for a Single Instance*, on page 454.

This operation can affect the ownership of the data instance. If the `DataWriter` was the exclusive owner of the instance, calling this operation will release that ownership, meaning ownership may be transferred to another, possibly lower strength, `DataWriter`.

The operation must be called only on registered instances. Otherwise the operation returns the error `RETCODE_PRECONDITION_NOT_MET`.

Instance Handle

The `HANDLE_NIL` handle value can be used for the parameter handle. This indicates that the identity of the instance is automatically deduced from the `instance_data` (by means of the key).

If handle is any value other than `HANDLE_NIL`, then it must correspond to the value returned by `register_instance` or `register_instance_w_timestamp` when the instance (identified by its key) was registered. If there is no correspondence, the result of the operation is unspecified.

The sample that is passed as `instance_data` is only used to check for consistency between its key values and the supplied `instance_handle`: the sample itself will not actually be delivered to the connected `DataReaders`.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `unregister_instance` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. In case the synchronous attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `unregister_instance` operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the `unregister_instance` operation will fail and returns `RETCODE_TIMEOUT`.

Sample Validation

OpenSplice DDS offers the possibility to check the sample that is passed as `instance_data` for validity. Because validity checking might reduce the overall performance, it is by default disabled. This has been done by enclosing the validity checking with conditional compiler directives like this:

```
#ifdef OSPL_BOUNDS_CHECK
```



```
        // check a specific bound.
    #endif
```

i

By defining a macro called `OSPL_OSPL_BOUNDS_CHECK`, the validity checking will be included. On most compilers this macro can be defined by passing an additional command line parameter called `-DOSPL_BOUNDS_CHECK`.

Since the `unregister_instance` operation merely uses the sample to check for consistency between its key values and the supplied `instance_handle`, only these keyfields will be validated against the restrictions imposed by the IDL to C++ language mapping, where:

- an enum may not exceed the value of its highest label
- a string (bounded or unbounded) may not be `NULL`. (Use `"` for an empty string instead)
- the length of a bounded string may not exceed the limit specified in IDL

If any of these restrictions is violated when validity checking is enabled, the operation will fail and return a `RETCODE_BAD_PARAMETER`. More specific information about the context of this error will be written to the error log. When validity checking is disabled, any of these violations may result in undefined behaviour.

Return Code

When the operation returns:

- `RETCODE_OK` - the Data Distribution Service is informed that the instance will not be modified any more
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - handle is not a valid handle or `instance_data` is not a valid sample.
- `RETCODE_ALREADY_DELETED` - the `FooDataWriter` has already been deleted
- `RETCODE_NOT_ENABLED` - the `FooDataWriter` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - the handle has not been registered with this `FooDataWriter`.
- `RETCODE_TIMEOUT` - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `unregister_instance` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.55 unregister_instance_w_timestamp

Scope

SPACE::FooDataWriter

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
        unregister_instance_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t handle,
             const Time_t& source_timestamp);
```

Description

This operation will inform the Data Distribution Service that the application will **not** be modifying a particular instance any more and provides a value for the `source_timestamp` explicitly.

Parameters

in Foo instance_data - the instance to which the application was writing or disposing.

in InstanceHandle_t handle - the handle to the instance, which has been used for writing and disposing.

in const Time_t& source_timestamp - the timestamp used.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation performs the same functions as `unregister_instance` except that the application provides the value for the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Return Code

When the operation returns:

- `RETCODE_OK` - the Data Distribution Service is informed that the instance will not be modified any more

- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - handle is not a valid handle or instance_data is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this `FooDataWriter`.
- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `unregister_instance_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.56 write

Scope

`SPACE::FooDataWriter`

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
    write
        (const Foo& instance_data,
         InstanceHandle_t handle);
```

Description

This operation modifies the value of a data instance.

Parameters

in `const Foo& instance_data` - the data to be written.

in `InstanceHandle_t handle` - the handle to the instance as supplied by `register_instance`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation modifies the value of a data instance. When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

As a side effect, this operation asserts liveness on the `DataWriter` itself and on the containing `DomainParticipant`.

Before writing data to an instance, the instance may be registered with the `register_instance` or `register_instance_w_timestamp` operation. The handle returned by one of the `register_instance` operations can be supplied to the parameter handle of the `write` operation. However, it is also possible to supply the special `HANDLE_NIL` handle value, which means, that the identity of the instance is automatically deduced from the `instance_data` (identified by the key).

Instance Handle

The `HANDLE_NIL` handle value can be used for the parameter handle. This indicates the identity of the instance is automatically deduced from the `instance_data` (by means of the key).

If handle is any value other than `HANDLE_NIL`, it must correspond to the value returned by `register_instance` or `register_instance_w_timestamp` when the instance (identified by its key) was registered. Passing such a registered handle helps the Data Distribution Service to process the sample more efficiently. If there is no correspondence between handle and sample, the result of the operation is unspecified.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `write` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, is exceeded. In case the `synchronous` attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the

`max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the write operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the write operation will fail and returns `RETCODE_TIMEOUT`.

Sample Validation

OpenSplice DDS offers the possibility to check the sample that is passed as `instance_data` for validity. Because validity checking might reduce the overall performance, it is by default disabled. This has been done by enclosing the validity checking with conditional compiler directives like this:

```
#ifdef OSPL_BOUNDS_CHECK
    // check a specific bound.
#endif
```

i

By defining a macro called `OSPL_OSPL_BOUNDS_CHECK`, the validity checking will be included. On most compilers this macro can be defined by passing an additional command line parameter called `-DOSPL_BOUNDS_CHECK`.

Before the sample is accepted by the `DataWriter`, it is validated against the restrictions imposed by the IDL to C++ language mapping, where:

- an enum may not exceed the value of its highest label
- a string (bounded or unbounded) may not be `NULL`. (Use `""` for an empty string instead)
- the length of a bounded string may not exceed the limit specified in IDL
- the length of a bounded sequence may not exceed the limit specified in IDL

If any of these restrictions is violated when validity checking is enabled, the operation will fail and return a `RETCODE_BAD_PARAMETER`. More specific information about the context of this error will be written to the error log. When validity checking is disabled, any of these violations may result in undefined behaviour.

Be aware that it is not possible for the middleware to determine whether a union is correctly initialized, since according to the IDL-C++ language mapping a union just returns its current contents in the format of the requested branch without performing any checks. It is therefore the responsibility of the application programmer to make sure that the requested branch actually corresponds to the currently active branch. Not doing so may result in undefined behaviour as well.

Return Code

When the operation returns:

- *RETCODE_OK* - the value of a data instance is modified
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_BAD_PARAMETER* - handle is not a valid handle or instance_data is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this `FooDataWriter`.
- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the write operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.57 write_w_timestamp

Scope

SPACE::FooDataWriter

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
        write_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t handle,
             const Time_t& source_timestamp);
```

Description

This operation modifies the value of a data instance and provides a value for the `source_timestamp` explicitly.

Parameters

in const Foo& instance_data - the data to be written.

in InstanceHandle_t handle - the handle to the instance as supplied by `register_instance`.

in const Time_t& source_timestamp - the timestamp used.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation performs the same functions as `write` except that the application provides the value for the parameter `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Return Code

When the operation returns:

- `RETCODE_OK` - the value of a data instance is modified
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_BAD_PARAMETER` - handle is not a valid handle or `instance_data` is not a valid sample.
- `RETCODE_ALREADY_DELETED` - the `FooDataWriter` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataWriter` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - the handle has not been registered with this `FooDataWriter`.
- `RETCODE_TIMEOUT` - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `write_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.58 `writedispose`

Scope

`SPACE::FooDataWriter`

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
```

```
writedispose
(const Foo& instance_data,
 InstanceHandle_t handle);
```

Description

This operation modifies and disposes a data instance.

Parameters

in const Foo& instance_data - the data to be written and disposed.

in InstanceHandle_t handle - the handle to the instance as supplied by `register_instance`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

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 `DataReader` and, dependent on the `QoSPolicy` settings, also in the Transient and Persistent stores, will be modified and marked for deletion by setting their `InstanceStateKind` to `NOT_ALIVE_DISPOSED_INSTANCE_STATE`.

When this operation is used, the Data Distribution Service will automatically supply the value of the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQoSPolicy`.

As a side effect, this operation asserts liveness on the `DataWriter` itself and on the containing `DomainParticipant`.

Effects on DataReaders

Actual deletion of the instance administration in a connected `DataReader` will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it.
- A `DataWriter` can register an instance explicitly by using one of the special operations `register_instance` or `register_instance_w_timestamp`.

- A `DataWriter` can register an instance implicitly by using the special constant `HANDLE_NIL` in any of the other `DataWriter` operations.
- A `DataWriter` can unregister an instance explicitly by using one of the special operations `unregister_instance` or `unregister_instance_w_timestamp`.
- A `DataWriter` will unregister all its contained instances implicitly when it is deleted.
- When a `DataReader` detects a loss of liveliness in one of its connected `DataWriters`, it will consider all instances registered by that `DataWriter` as being implicitly unregistered.
- **and** the application must have consumed all samples belonging to the instance, either implicitly or explicitly.
 - An application can consume samples explicitly by invoking the `take` operation, or one of its variants, on its `DataReaders`.
 - The `DataReader` can consume disposed samples implicitly when the `autopurge_disposed_samples_delay` of the `ReaderDataLifecycleQosPolicy` has expired.

The `DataReader` may also remove instances that haven't been disposed first: this happens when the `autopurge_nowriter_samples_delay` of the `ReaderDataLifecycleQosPolicy` has expired after the instance is considered unregistered by all connected `DataWriters` (i.e. when it has a `InstanceStateKind` of `NOT_ALIVE_NO_WRITERS`). See also Section 3.1.3.15, *ReaderDataLifecycleQosPolicy*, on page 65.

Effects on Transient/Persistent Stores

Actual deletion of the instance administration in the connected Transient and Persistent stores will be postponed until the following conditions have been met:

- the instance must be unregistered (either implicitly or explicitly) by all connected `DataWriters` that have previously registered it. (See above.)
- **and** the period of time specified by the `service_cleanup_delay` attribute in the `DurabilityServiceQosPolicy` on the `Topic` must have elapsed after the instance is considered unregistered by all connected `DataWriters`.

See also Section 3.1.3.4, *DurabilityServiceQosPolicy*, on page 48.

Instance Handle

The `HANDLE_NIL` handle value can be used for the parameter `handle`. This indicates the identity of the instance is automatically deduced from the `instance_data` (by means of the key).

If `handle` is any value other than `HANDLE_NIL`, it must correspond to the value that was returned by either the `register_instance` operation or the `register_instance_w_timestamp` operation, when the instance (identified by its key) was registered. If there is no correspondence, the result of the operation is unspecified.

The sample that is passed as `instance_data` will actually be delivered to the connected `DataReaders`, but will immediately be marked for deletion.

Blocking

If the `HistoryQosPolicy` is set to `KEEP_ALL_HISTORY_QOS`, the `writediscard` operation on the `DataWriter` may block if the modification would cause data to be lost because one of the limits, specified in the `ResourceLimitsQosPolicy`, to be exceeded. In case the `synchronous` attribute value of the `ReliabilityQosPolicy` is set to `TRUE` for communicating `DataWriters` and `DataReaders` then the `DataWriter` will wait until all synchronous `DataReaders` have acknowledged the data. Under these circumstances, the `max_blocking_time` attribute of the `ReliabilityQosPolicy` configures the maximum time the `writediscard` operation may block (either waiting for space to become available or data to be acknowledged). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits and all expected acknowledgements are received, the `writediscard` operation will fail and returns `RETCODE_TIMEOUT`.

Sample Validation

OpenSplice DDS offers the possibility to check the sample that is passed as `instance_data` for validity. Because validity checking might reduce the overall performance, it is by default disabled. This has been done by enclosing the validity checking with conditional compiler directives like this:

```
#ifdef OSPL_BOUNDS_CHECK
    // check a specific bound.
#endif
```

i

By defining a macro called `OSPL_BOUNDS_CHECK`, the validity checking will be included. On most compilers this macro can be defined by passing an additional command line parameter called `-DOSPL_BOUNDS_CHECK`.

Before the sample is accepted by the `DataWriter`, it is validated against the restrictions imposed by the IDL to C++ language mapping, where:

- an enum may not exceed the value of its highest label
- a string (bounded or unbounded) may not be `NULL`. (Use `""` for an empty string instead)

- the length of a bounded string may not exceed the limit specified in IDL
- the length of a bounded sequence may not exceed the limit specified in IDL

If any of these restrictions is violated when validity checking is enabled, the operation will fail and return a `RETCODE_BAD_PARAMETER`. More specific information about the context of this error will be written to the error log. When validity checking is disabled, any of these violations may result in undefined behaviour.

Be aware that it is not possible for the middleware to determine whether a union is correctly initialized, since according to the IDL-C++ language mapping a union just returns its current contents in the format of the requested branch without performing any checks. It is therefore the responsibility of the application programmer to make sure that the requested branch actually corresponds to the currently active branch. Not doing so may result in undefined behaviour as well.

Return Code

When the operation returns:

- `RETCODE_OK` - the Data Distribution Service has modified the instance and marked it for deletion.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - handle is not a valid handle or instance_data is not a valid sample.
- `RETCODE_ALREADY_DELETED` - the `FooDataWriter` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataWriter` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - the handle has not been registered with this `SPACE_FooDataWriter`.
- `RETCODE_TIMEOUT` - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `writedispose` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.2.59 writedispose_w_timestamp

Scope

`SPACE::FooDataWriter`

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
        writedispose_w_timestamp
            (const Foo& instance_data,
             InstanceHandle_t handle,
             const Time_t& source_timestamp);
```

Description

This operation requests the Data Distribution Service to modify the instance and mark it for deletion, and provides a value for the `source_timestamp` explicitly.

Parameters

in const Foo& instance_data - the data to be written and disposed.

in InstanceHandle_t handle - the handle to the instance as supplied by `register_instance`.

in const Time_t& source_timestamp - the timestamp used.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_TIMEOUT`.

Detailed Description

This operation performs the same functions as `writedispose` except that the application provides the value for the `source_timestamp` that is made available to connected `DataReader` objects. This timestamp is important for the interpretation of the `DestinationOrderQosPolicy`.

Return Code

When the operation returns:

- *RETCODE_OK* - the Data Distribution Service has modified the instance and marked it for deletion.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - `handle` is not a valid handle or `instance_data` is not a valid sample.
- *RETCODE_ALREADY_DELETED* - the `FooDataWriter` has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

- *RETCODE_NOT_ENABLED* - the `FooDataWriter` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - the handle has not been registered with this `SPACE_FooDataWriter`.
- *RETCODE_TIMEOUT* - either the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`, or the current action was waiting for data delivery acknowledgement by synchronous `DataReaders`. This caused blocking of the `writedispose_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

3.4.3 PublisherListener Interface

Since a `Publisher` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `PublisherListener`. This interface must be implemented by the application. A user defined class must be provided by the application which must extend from the `PublisherListener` class. **All** `PublisherListener` operations **must** be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.



All operations for this interface must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

The `PublisherListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `PublisherListener` is related to changes in communication status.

The interface description of this class is as follows:

```
class PublisherListener {
//
// inherited from DataWriterListener
//
// void
//   on_offered_deadline_missed
//     (DataWriter_ptr writer,
//      const OfferedDeadlineMissedStatus& status) = 0;

// void
//   on_offered_incompatible_qos
//     (DataWriter_ptr writer,
//      const OfferedIncompatibleQosStatus& status) = 0;
```

```

// void
//     on_liveliness_lost
//         (DataWriter_ptr writer,
//         const LivelinessLostStatus& status) = 0;

// void
//     on_publication_matched
//         (DataWriter_ptr writer,
//         const PublicationMatchedStatus& status) = 0;
//
// implemented API operations
//     <no operations>
//
};

```

The next paragraphs list all `PublisherListener` operations. Since these operations are all inherited, they are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.4.3.1 `on_liveliness_lost` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_liveliness_lost
        (DataWriter_ptr writer,
         const LivelinessLostStatus& status) = 0;

```

3.4.3.2 `on_offered_deadline_missed` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_offered_deadline_missed
        (DataWriter_ptr writer,
         const OfferedDeadlineMissedStatus& status) = 0;

```

3.4.3.3 `on_offered_incompatible_qos` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_offered_incompatible_qos
        (DataWriter_ptr writer,
         const OfferedIncompatibleQosStatus& status) = 0;
```

3.4.3.4 on_publication_matched (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataWriterListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_publication_matched
        (DataWriter_ptr writer,
         const PublicationMatchedStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.4.4 DataWriterListener Interface

Since a `DataWriter` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `DataWriterListener`. This interface must be implemented by the application. A user defined class must be provided by the application which must extend from the `DataWriterListener` class. **All** `DataWriterListener` operations **must** be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.



All operations for this interface must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

The `DataWriterListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `DataWriterListener` is related to changes in communication status.

The interface description of this class is as follows:

```
class DataWriterListener
{
    // abstract external operations
    void
        on_offered_deadline_missed
            (DataWriter_ptr writer,
             const OfferedDeadlineMissedStatus& status) = 0;
```

```

void
    on_offered_incompatible_qos
        (DataWriter_ptr writer,
         const OfferedIncompatibleQosStatus& status) = 0;

void
    on_liveliness_lost
        (DataWriter_ptr writer,
         const LivelinessLostStatus& status) = 0;

void
    on_publication_matched
        (DataWriter_ptr writer,
         const PublicationMatchedStatus& status) = 0;
// implemented API operations
// <no operations>
};

```

The next paragraphs describe the usage of all `DataWriterListener` operations. These abstract operations are fully described because they must be implemented by the application.

3.4.4.1 `on_liveliness_lost` (abstract)

Scope

`DDS::DataWriterListener`

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_liveliness_lost
        (DataWriter_ptr writer,
         const LivelinessLostStatus& status) = 0;

```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `LivelinessLostStatus` changes.

Parameters

in `DataWriter_ptr writer` - contains a pointer to the `DataWriter` on which the `LivelinessLostStatus` has changed (this is an input to the application).
in `const LivelinessLostStatus& status` - contains the `LivelinessLostStatus` struct (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `LivelinessLostStatus` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataWriterListener` is installed and enabled for the liveliness lost status. The liveliness lost status will change when the liveliness that the `DataWriter` has committed through its `LivelinessQosPolicy` was not respected. In other words, the `DataWriter` failed to actively signal its liveliness within the offered liveliness period. As a result, the `DataReader` objects will consider the `DataWriter` as no longer “alive”.

The Data Distribution Service will call the `DataWriterListener` operation with a parameter `writer`, which will contain a reference to the `DataWriter` on which the conflict occurred and a parameter `status`, which will contain the `LivelinessLostStatus` struct.

3.4.4.2 on_offered_deadline_missed (abstract)

Scope

`DDS::DataWriterListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_offered_deadline_missed
        (DataWriter_ptr writer,
         const OfferedDeadlineMissedStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `OfferedDeadlineMissedStatus` changes.

Parameters

in DataWriter_ptr writer - contain a pointer to the `DataWriter` on which the `OfferedDeadlineMissedStatus` has changed (this is an input to the application).

in const OfferedDeadlineMissedStatus& status - contain the `OfferedDeadlineMissedStatus` struct (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `OfferedDeadlineMissedStatus` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataWriterListener` is installed and enabled for the offered deadline missed status. The offered deadline missed status will change when the deadline that the `DataWriter` has committed through its `DeadlineQosPolicy` was not respected for a specific instance.

The Data Distribution Service will call the `DataWriterListener` operation with a parameter `writer`, which will contain a reference to the `DataWriter` on which the conflict occurred and a parameter `status`, which will contain the `OfferedDeadlineMissedStatus` struct.

3.4.4.3 on_offered_incompatible_qos (abstract)

Scope

`DDS::DataWriterListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_offered_incompatible_qos
        (DataWriter_ptr writer,
         const OfferedIncompatibleQosStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `OFFERED_INCOMPATIBLE_QOS_STATUS` changes.

Parameters

in DataWriter_ptr writer - contain a pointer to the `DataWriter` on which the `OFFERED_INCOMPATIBLE_QOS_STATUS` has changed (this is an input to the application).

in const OfferedIncompatibleQosStatus& status - contain the `OfferedIncompatibleQosStatus` struct (this is an input to the application).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `OFFERED_INCOMPATIBLE_QOS_STATUS` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataWriterListener` is installed and enabled for the `OFFERED_INCOMPATIBLE_QOS_STATUS`. The incompatible Qos status will change when a `DataReader` object has been discovered by the `DataWriter` with the same `Topic` and a requested `DataReaderQos` that was incompatible with the one offered by the `DataWriter`.

The Data Distribution Service will call the `DataWriterListener` operation with a parameter `writer`, which will contain a reference to the `DataWriter` on which the conflict occurred and a parameter `status`, which will contain the `OfferedIncompatibleQosStatus` struct.

3.4.4.4 `on_publication_matched` (abstract)

Scope

`DDS::DataWriterListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_publication_matched
        (DataWriter_ptr writer,
         const PublicationMatchedStatus& status) = 0;
```

Description

This operation is not yet supported. It is scheduled for a future release.

3.5 Subscription Module

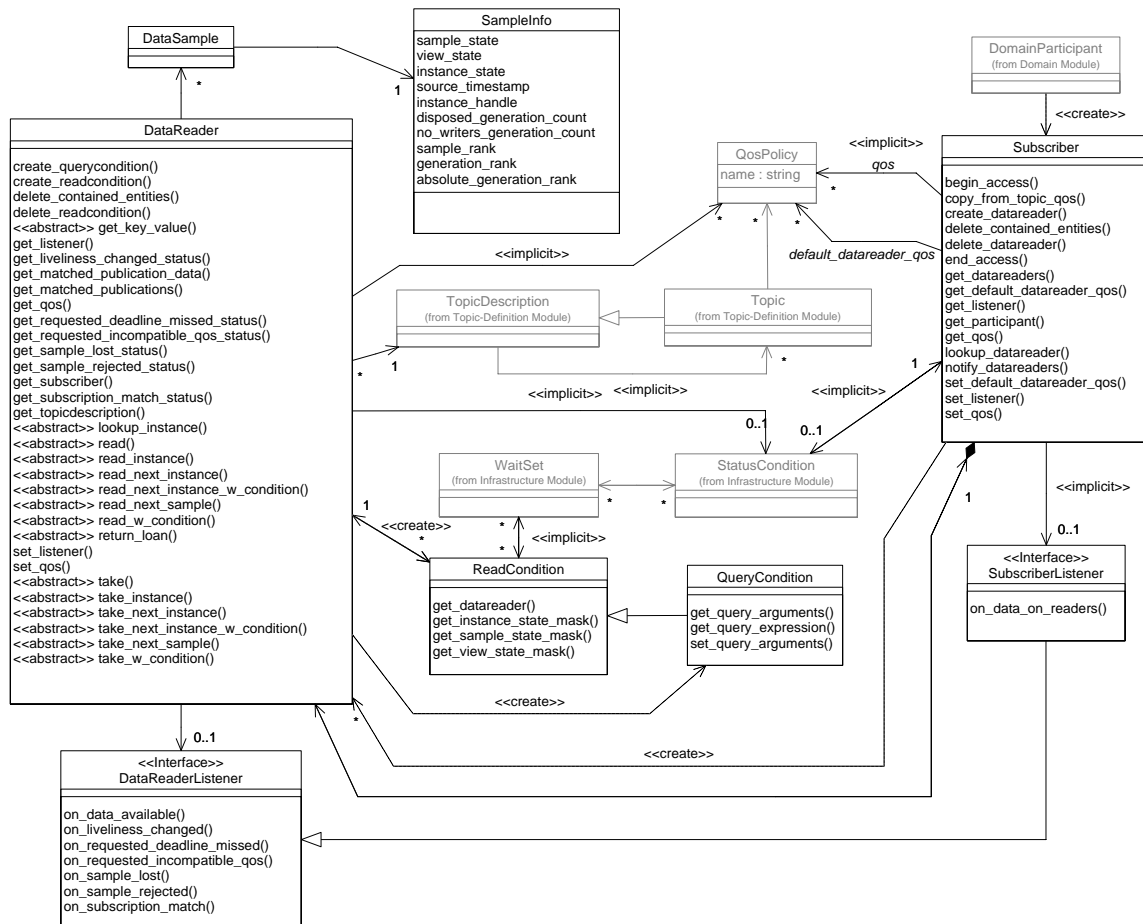


Figure 19 DCPS Subscription Module's Class Model

This module contains the following classes:

- Subscriber
- **Subscription type specific classes**
- DataSample
- SampleInfo (struct)
- SubscriberListener (interface)
- DataReaderListener (interface)
- ReadCondition
- QueryCondition.

“Subscription type specific classes” contains the generic class and the generated data type specific classes. For each data type, a data type specific class `<type>DataReader` is generated (based on IDL) by calling the pre-processor.

For instance, for the fictional data type `Foo` (this also applies to other types) “Subscription type specific classes” contains the following classes:

- `DataReader` (abstract)
- `FooDataReader`.

A `Subscriber` is an object responsible for receiving published data and making it available (according to the `SubscriberQos`) to the application. It may receive and dispatch `Topic` with data of different specified data types. To access the received data, the application must use a typed `DataReader` attached to the `Subscriber`. Thus, a subscription is defined by the association of a `DataReader` with a `Subscriber`. This association expresses the intent of the application to subscribe to the data described by the `DataReader` in the context provided by the `Subscriber`.

3.5.1 Class Subscriber

A `Subscriber` is the object responsible for the actual reception of the data resulting from its subscriptions.

A `Subscriber` acts on behalf of one or more `DataReader` objects that are related to it. When it receives data (from the other parts of the system), it indicates to the application that data is available through its `DataReaderListener` and by enabling related `Conditions`. The application can access the list of concerned `DataReader` objects through the operation `get_datareaders` and then access the data available through operations on the `DataReader`.

The interface description of this class is as follows:

```
class Subscriber
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//   (void);
// StatusMask
//   get_status_changes
//   (void);
// ReturnCode_t
//   enable
//   (void);
//
// implemented API operations
//
    DataReader_ptr
```

```
create_datareader
    (TopicDescription_ptr a_topic,
     const DataReaderQos& qos,
     DataReaderListener_ptr a_listener,
     StatusMask mask);

ReturnCode_t
delete_datareader
    (DataReader_ptr a_datareader);

ReturnCode_t
delete_contained_entities
    (void);

DataReader_ptr
lookup_datareader
    (const char* topic_name);

ReturnCode_t
get_datareaders
    (DataReaderSeq& readers,
     SampleStateMask sample_states,
     ViewStateMask view_states,
     InstanceStateMask instance_states);

ReturnCode_t
notify_datareaders
    (void);

ReturnCode_t
set_qos
    (const SubscriberQos& qos);

ReturnCode_t
get_qos
    (SubscriberQos& qos);

ReturnCode_t
set_listener
    (SubscriberListener_ptr a_listener,
     StatusMask mask);

SubscriberListener_ptr
get_listener
    (void);

ReturnCode_t
begin_access
    (void);

ReturnCode_t
```

```

        end_access
        (void);

DomainParticipant_ptr
get_participant
        (void);

ReturnCode_t
set_default_datareader_qos
        (const DataReaderQos& qos);

ReturnCode_t
get_default_datareader_qos
        (DataReaderQos& qos);

ReturnCode_t
copy_from_topic_qos
        (DataReaderQos& a_datareader_qos,
         const TopicQos& a_topic_qos);
};

```

The next paragraphs describe the usage of all `Subscriber` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.1.1 `begin_access`

Scope

DDS::Subscriber

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
begin_access
        (void);

```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.1.2 `copy_from_topic_qos`

Scope

DDS::Subscriber

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
copy_from_topic_qos
        (DataReaderQos& a_datareader_qos,

```

```
const TopicQos& a_topic_qos);
```

Description

This operation will copy the policies in `a_topic_qos` to the corresponding policies in `a_datareader_qos`.

Parameters

inout DataReaderQos& a_datareader_qos - the destination DataReaderQos struct to which the QosPolicy settings will be copied.

in const TopicQos& a_topic_qos - the source TopicQos, which will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation will copy the QosPolicy settings in `a_topic_qos` to the corresponding QosPolicy settings in `a_datareader_qos` (replacing the values in `a_datareader_qos`, if present).

This is a “convenience” operation, useful in combination with the operations `get_default_datawriter_qos` and `Topic::get_qos`. The operation `copy_from_topic_qos` can be used to merge the DataReader default QosPolicy settings with the corresponding ones on the Topic. The resulting DataReaderQos can then be used to create a new DataReader, or set its DataReaderQos.

This operation does not check the resulting `a_datareader_qos` for self consistency. This is because the “merged” `a_datareader_qos` may not be the final one, as the application can still modify some QosPolicy settings prior to applying the DataReaderQos to the DataReader.

Return Code

When the operation returns:

- `RETCODE_OK` - the QosPolicy settings have successfully been copied from the TopicQos to the DataReaderQos
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the Subscriber has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.3 create_datareader

Scope

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
DataReader_ptr
create_datareader
    (TopicDescription_ptr a_topic,
     const DataReaderQos& qos,
     DataReaderListener_ptr a_listener,
     StatusMask mask);
```

Description

This operation creates a DataReader with the desired QosPolicy settings, for the desired TopicDescription and attaches the optionally specified DataWriterListener to it.

Parameters

- in TopicDescription_ptr a_topic* - a pointer to the TopicDescription for which the DataReader is created. This may be a Topic, MultiTopic or ContentFilteredTopic.
- in const DataReaderQos& qos* - the struct with the QosPolicy settings for the new DataReader, when these QosPolicy settings are not self consistent, no DataReader is created.
- in DataReaderListener_ptr a_listener* - a pointer to the DataReaderListener instance which will be attached to the new DataReader. It is permitted to use NULL as the value of the listener: this behaves as a DataWriterListener whose operations perform no action.
- in StatusMask mask* - a bit-mask in which each bit enables the invocation of the DataReaderListener for a certain status.

Return Value

DataReader_ptr - a pointer to the newly created DataReader. In case of an error, the NULL pointer is returned.

Detailed Description

This operation creates a DataReader with the desired QosPolicy settings, for the desired TopicDescription and attaches the optionally specified DataReaderListener to it. The TopicDescription may be a Topic, MultiTopic or ContentFilteredTopic. The returned DataReader is attached

(and belongs) to the Subscriber. To delete the `DataReader` the operation `delete_datareader` or `delete_contained_entities` must be used. If no read rights are defined for the specific topic then the creation of the `DataReader` will fail.

Application Data Type

The `DataReader` returned by this operation is an object of a derived class, specific to the data type associated with the `TopicDescription`. For each application-defined data type `<type>` there is a class `<type>DataReader` generated by calling the pre-processor. This data type specific class extends `DataReader` and contains the operations to read data of data type `<type>`.

Because the `DataReader` may read a `Topic`, `ContentFilteredTopic` or `MultiTopic`, the `DataReader` is associated with the `TopicDescription`. The `DataWriter` can only write a `Topic`, **not** a `ContentFilteredTopic` or `MultiTopic`, because these two are constructed at the Subscriber side.

QosPolicy

The common application pattern to construct the `QosPolicy` settings for the `DataReader` is to:

- Retrieve the `QosPolicy` settings on the associated `TopicDescription` by means of the `get_qos` operation on the `TopicDescription`
- Retrieve the default `DataReaderQos` by means of the `get_default_datareader_qos` operation on the Subscriber
- Combine those two `QosPolicy` settings and selectively modify policies as desired (`copy_from_topic_qos`)
- Use the resulting `QosPolicy` settings to construct the `DataReader`.
- In case the specified `QosPolicy` settings are not self consistent, no `DataReader` is created and the `NULL` pointer is returned.

Default QoS

The constant `DATAREADER_QOS_DEFAULT` can be used as parameter `qos` to create a `DataReader` with the default `DataReaderQos` as set in the Subscriber. The effect of using `DATAREADER_QOS_DEFAULT` is the same as calling the operation `get_default_datareader_qos` and using the resulting `DataReaderQos` to create the `DataReader`.

The special `DATAREADER_QOS_USE_TOPIC_QOS` can be used to create a `DataReader` with a combination of the default `DataReaderQos` and the `TopicQos`. The effect of using `DATAREADER_QOS_USE_TOPIC_QOS` is the same as calling the operation `get_default_datareader_qos` and retrieving the `TopicQos` (by means of the operation `Topic::get_qos`) and then combining

these two `QosPolicy` settings using the operation `copy_from_topic_qos`, whereby any common policy that is set on the `TopicQos` “overrides” the corresponding policy on the default `DataReaderQos`. The resulting `DataReaderQos` is then applied to create the `DataReader`.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `DataReaderListener` operation 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 fact that the status is reset prior to calling the listener means that if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

The following statuses are applicable to the `DataReaderListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_CHANGED_STATUS`
- `SUBSCRIPTION_MATCHED_STATUS`.

Status bits are declared as a constant and can be used by the application in an `OR` operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

In case a communication status is not activated in the mask of the `DataReaderListener`, the `SubscriberListener` of the containing `Subscriber` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `SubscriberListener` of the containing `Subscriber` and a `DataReader` specific behaviour when needed. In case the communication status is not activated in the mask of the `SubscriberListener` as well, the communication status will be propagated to the `DomainParticipantListener` of the containing `DomainParticipant`. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

3.5.1.4 delete_contained_entities

Scope

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_contained_entities
        (void);
```

Description

This operation deletes all the `DataReader` objects that were created by means of the `create_datareader` operation on the `Subscriber`.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation deletes all the `DataReader` objects that were created by means of the `create_datareader` operation on the `Subscriber`. In other words, it deletes all contained `DataReader` objects. Prior to deleting each `DataReader`, this operation recursively calls the corresponding `delete_contained_entities` operation on each `DataReader`. In other words, all `DataReader` objects in the `Subscriber` are deleted, including the `QueryCondition` and `ReadCondition` objects contained by the `DataReader`.



Note: The operation will return `PRECONDITION_NOT_MET` if the any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained `DataReader` cannot be deleted because the application has called a `read` or `take` operation and has not called the corresponding `return_loan` operation to return the loaned samples. In such cases, the operation does not roll-back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

- *RETCODE_OK* - the contained Entity objects are deleted and the application may delete the Subscriber;
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - one or more of the contained entities are in a state where they cannot be deleted.

3.5.1.5 delete_datareader

Scope

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_datareader
        (DataReader_ptr a_datareader);
```

Description

This operation deletes a DataReader that belongs to the Subscriber.

Parameters

in DataReader_ptr a_datareader - a pointer to the DataReader, which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES* or *RETCODE_PRECONDITION_NOT_MET*.

Detailed Description

This operation deletes a DataReader that belongs to the Subscriber. When the operation is called on a different Subscriber, as used when the DataReader was created, the operation has no effect and returns *RETCODE_PRECONDITION_NOT_MET*. The deletion of the DataReader is not allowed if there are any ReadCondition or QueryCondition objects that are attached to the DataReader. In that case the operation returns *RETCODE_PRECONDITION_NOT_MET*.

Return Code

When the operation returns:

- *RETCODE_OK* - the DataReader is deleted
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter `a_datareader` is not a valid `DataReader_ptr`
- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different Subscriber, as used when the DataReader was created, or the DataReader contains one or more `ReadCondition` or `QueryCondition` objects.

3.5.1.6 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    enable
        (void);
```

3.5.1.7 end_access**Scope**

`DDS::Subscriber`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    end_access
        (void);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.1.8 get_datareaders**Scope**

`DDS::Subscriber`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_datareaders
        (DataReaderSeq& readers,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.1.9 get_default_datareader_qos

Scope

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_default_datareader_qos
        (DataReaderQos& qos);
```

Description

This operation gets the default QosPolicy settings of the DataReader.

Parameters

inout DataReaderQos& qos - a reference to the DataReaderQos struct (provided by the application) in which the default QosPolicy settings for the DataReader are written.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation gets the default QosPolicy settings of the DataReader (that is the DataReaderQos) which is used for newly created DataReader objects, in case the constant DATAREADER_QOS_DEFAULT is used. The default DataReaderQos is only used when the constant is supplied as parameter qos to specify the DataReaderQos in the create_datareader operation. The application must provide the DataReaderQos struct in which the QosPolicy settings can be stored

and pass the `qos` reference to the operation. The operation writes the default `QosPolicy` settings to the struct referenced to by `qos`. Any settings in the struct are overwritten.

The values retrieved by this operation match the values specified on the last successful call to `set_default_datareader_qos`, or, if the call was never made, the default values as specified for each `QosPolicy` setting as defined in Table 2 on page 37.

Return Code

When the operation returns:

- `RETCODE_OK` - the default `DataReader QosPolicy` settings of this `Subscriber` have successfully been copied into the specified `DataReaderQos` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `Subscriber` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the `Data Distribution Service` ran out of resources to complete this operation.

3.5.1.10 `get_listener`

Scope

`DDS::Subscriber`

Synopsis

```
#include <ccpp_dds_dcps.h>
SubscriberListener_ptr
    get_listener
        (void);
```

Description

This operation allows access to a `SubscriberListener`.

Parameters

<none>

Return Value

`SubscriberListener_ptr` - result is a pointer to the `SubscriberListener` attached to the `Subscriber`.

Detailed Description

This operation allows access to a `SubscriberListener` attached to the `Subscriber`. When no `SubscriberListener` was attached to the `Subscriber`, the `NULL` pointer is returned.

3.5.1.11 `get_participant`

Scope

`DDS::Subscriber`

Synopsis

```
#include <ccpp_dds_dcps.h>
DomainParticipant_ptr
    get_participant
        (void);
```

Description

This operation returns the `DomainParticipant` associated with the `Subscriber` or the `NULL` pointer.

Parameters

<none>

Return Value

DomainParticipant_ptr - a pointer to the `DomainParticipant` associated with the `Subscriber` or the `NULL` pointer.

Detailed Description

This operation returns the `DomainParticipant` associated with the `Subscriber`. Note that there is exactly one `DomainParticipant` associated with each `Subscriber`. When the `Subscriber` was already deleted (there is no associated `DomainParticipant` any more), the `NULL` pointer is returned.

3.5.1.12 `get_qos`

Scope

`DDS::Subscriber`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (SubscriberQos& qos);
```

Description

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

Parameters

inout SubscriberQos& qos - a reference to the destination SubscriberQos struct in which the QosPolicy settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation allows access to the existing set of QoS policies of a Subscriber on which this operation is used. This SubscriberQos is stored at the location pointed to by the qos parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoS policy values applied to this Subscriber has successfully been copied into the specified SubscriberQos parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.13 **get_status_changes (inherited)**

This operation is inherited and therefore not described here. See the class Entity for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_status_changes
        (void);
```

3.5.1.14 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class Entity for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

3.5.1.15 lookup_datareader**Scope**

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
DataReader_ptr
    lookup_datareader
        (const char* topic_name);
```

Description

This operation returns a previously created `DataReader` belonging to the `Subscriber` which is attached to a `Topic` with the matching `topic_name`.

Parameters

in `const char* topic_name` - the name of the `Topic`, which is attached to the `DataReader` to look for.

Return Value

`DataReader_ptr` - Return value is a reference to the `DataReader` found. When no such `DataReader` is found, the `NULL` pointer is returned.

Detailed Description

This operation returns a previously created `DataReader` belonging to the `Subscriber` which is attached to a `Topic` with the matching `topic_name`. When multiple `DataReader` objects (which satisfy the same condition) exist, this operation will return one of them. It is not specified which one.

This operation may be used on the built-in `Subscriber`, which returns the built-in `DataReader` objects for the built-in `Topics`.

3.5.1.16 notify_datareaders**Scope**

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    notify_datareaders
        (void);
```

Description

This operation invokes the `on_data_available` operation on `DataReaderListener` objects which are attached to the contained `DataReader` entities having new, available data.

Parameters

<none>

Return Value

DDS_ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `DDS_RETCODE_ALREADY_DELETED` or `DDS_RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation invokes the `on_data_available` operation on the `DataReaderListener` objects attached to contained `DataReader` entities that have received information, but which have not yet been processed by those `DataReaders`.

The `notify_datareaders` operation ignores the bit mask value of individual `DataReaderListener` objects, even when the `DATA_AVAILABLE_STATUS` bit has not been set on a `DataReader` that has new, available data. The `on_data_available` operation will still be invoked, when the `DATA_AVAILABLE_STATUS` bit has not been set on a `DataReader`, but will not propagate to the `DomainParticipantListener`.

When the `DataReader` has attached a `NULL` listener, the event will be consumed and will not propagate to the `DomainParticipantListener`. (Remember that a `NULL` listener is regarded as a listener that handles all its events as a `NOOP`).

Return Code

When the operation returns:

- *RETCODE_OK* - all appropriate listeners have been invoked
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the `Subscriber` has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - there are insufficient Data Distribution Service resources to complete this operation

3.5.1.17 `set_default_datareader_qos`

Scope

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_default_datareader_qos
        (const DataReaderQos& qos);
```

Description

This operation sets the default `DataReaderQos` of the `DataReader`.

Parameters

in `const DataReaderQos& qos` - the `DataReaderQos` struct, which contains the new default `QosPolicy` settings for the newly created `DataReaders`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_INCONSISTENT_POLICY`.

Detailed Description

This operation sets the default `DataReaderQos` of the `DataReader` (that is the struct with the `QosPolicy` settings). This `QosPolicy` is used for newly created `DataReader` objects in case the constant `DATAREADER_QOS_DEFAULT` is used as parameter `qos` to specify the `DataReaderQos` in the `create_datareader` operation. This operation checks if the `DataReaderQos` is self consistent. If it is not, the operation has no effect and returns `RETCODE_INCONSISTENT_POLICY`.

The values set by this operation are returned by `get_default_datareader_qos`.

Return Code

When the operation returns:

- *RETCODE_OK* - the new default `DataReaderQos` is set
- *RETCODE_ERROR* - an internal error has occurred.

- *RETCODE_BAD_PARAMETER* - the parameter *qos* is not a valid *DataReaderQos*. It contains a *QosPolicy* setting with an invalid *Duration_t* value or an enum value that is outside its legal boundaries.
- *RETCODE_UNSUPPORTED* - one or more of the selected *QosPolicy* values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the *Subscriber* has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the *Data Distribution Service* ran out of resources to complete this operation.
- *RETCODE_INCONSISTENT_POLICY* - the parameter *qos* contains conflicting *QosPolicy* settings, e.g. a history depth that is higher than the specified resource limits.

3.5.1.18 set_listener

Scope

DDS::Subscriber

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_listener
        (SubscriberListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation attaches a *SubscriberListener* to the *Subscriber*.

Parameters

in SubscriberListener_ptr a_listener - a pointer to the *SubscriberListener* instance, which will be attached to the *Subscriber*.

in StatusMask mask - a bit mask in which each bit enables the invocation of the *SubscriberListener* for a certain status.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_ALREADY_DELETED* or *RETCODE_OUT_OF_RESOURCES*.

Detailed Description

This operation attaches a `SubscriberListener` to the `Subscriber`. Only one `SubscriberListener` can be attached to each `Subscriber`. If a `SubscriberListener` was already attached, the operation will replace it with the new one. When a `_listener` is the `NULL` pointer, it represents a listener that is treated as a NOOP¹ for all statuses activated in the bit mask.

Communication Status

For each communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever that communication status changes. For each communication status activated in the mask, the associated `SubscriberListener` operation 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>_status` from inside the listener it will see the status already reset. An exception to this rule is the `NULL` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `SubscriberListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS` *(propagated)*
- `REQUESTED_INCOMPATIBLE_QOS_STATUS` *(propagated)*
- `SAMPLE_LOST_STATUS` *(propagated)*
- `SAMPLE_REJECTED_STATUS` *(propagated)*
- `DATA_AVAILABLE_STATUS` *(propagated)*
- `LIVELINESS_CHANGED_STATUS` *(propagated)*
- `SUBSCRIPTION_MATCHED_STATUS` *(propagated)*
- `DATA_ON_READERS_STATUS`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

The Data Distribution Service will trigger the most specific and relevant `Listener`. In other words, in case a communication status is also activated on the `DataReaderListener` of a contained `DataReader`, the `DataReaderListener` on that contained `DataReader` is invoked instead of the `SubscriberListener`.

1. Short for **No-Operation**, an instruction that does nothing.

This means, that a status change on a contained `DataReader` only invokes the `SubscriberListener` if the contained `DataReader` itself does not handle the trigger event generated by the status change.

In case a communication status is not activated in the mask of the `SubscriberListener`, the `DomainParticipantListener` of the containing `DomainParticipant` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `DomainParticipantListener` of the containing `DomainParticipant` and a `Subscriber` specific behaviour when needed. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its mask, the application is not notified of the change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are “Read Communication Statuses” and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a `DataReader`, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

Return Code

When the operation returns:

- `RETCODE_OK` - the `SubscriberListener` is attached
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_ALREADY_DELETED` - the `Subscriber` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.1.19 `set_qos`

Scope

`DDS::Subscriber`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const SubscriberQos& qos);
```


Description

This operation replaces the existing set of `QosPolicy` settings for a Subscriber.

Parameters

in const SubscriberQos& qos - new set of `QosPolicy` settings for the Subscriber.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_IMMUTABLE_POLICY` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation replaces the existing set of `QosPolicy` settings for a Subscriber. The parameter `qos` contains the `QosPolicy` settings which is checked for self-consistency and mutability. When the application tries to change a `QosPolicy` setting for an enabled Subscriber, which can only be set before the Subscriber is enabled, the operation will fail and a `RETCODE_IMMUTABLE_POLICY` is returned. In other words, the application must provide the presently set `QosPolicy` settings in case of the immutable `QosPolicy` settings. Only the mutable `QosPolicy` settings can be changed. When `qos` contains conflicting `QosPolicy` settings (not self-consistent), the operation will fail and a `RETCODE_INCONSISTENT_POLICY` is returned.

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 `RETCODE_OK`). If one or more of the partitions in the QoS structure have insufficient access rights configured then the `set_qos` function will fail with a `RETCODE_PRECONDITION_NOT_MET` error code.

Return Code

When the operation returns:

- `RETCODE_OK` - the new `SubscriberQos` is set
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `qos` is not a valid `SubscriberQos`. It contains a `QosPolicy` setting with an enum value that is outside its legal boundaries.
- `RETCODE_UNSUPPORTED` - one or more of the selected `QosPolicy` values are currently not supported by OpenSplice.

- *RETCODE_ALREADY_DELETED* - the Subscriber has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter qos contains an immutable QosPolicy setting with a different value than set during enabling of the Subscriber.
- *RETCODE_PRECONDITION_NOT_MET* - returned when insufficient access rights exist for the partition(s) listed in the QoS structure.

3.5.2 Subscription Type Specific Classes

“Subscription type specific classes” contains the generic class and the generated data type specific classes. For each data type, a data type specific class `<type>DataReader` is generated (based on IDL) by calling the pre-processor. In case of data type `Foo` (this also applies to other types); “Subscription type specific classes” contains the following classes:

This paragraph describes the generic `DataReader` class and the derived application type specific `<type>DataReader` classes which together implement the application subscription interface. For each application type, used as `Topic` data type, the pre-processor generates a `<type>DataReader` class from an IDL type description. The `FooDataReader` class that would be generated by the pre-processor for a fictional type `Foo` describes the `<type>DataReader` classes.

3.5.2.1 Class `DataReader` (abstract)

A `DataReader` allows the application:

- to declare data it wishes to receive (i.e., make a subscription)
- to access data received by the associated Subscriber.

A `DataReader` refers to exactly one `TopicDescription` (either a `Topic`, a `ContentFilteredTopic` or a `MultiTopic`) that identifies the samples to be read. The `DataReader` may give access to several instances of the data type, which are distinguished from each other by their key.

`DataReader` is an abstract class. It is specialized for each particular application data type. For a fictional application data type “`Foo`” (defined in the module `SPACE`) the specialized class would be `SPACE::FooDataReader`.

The interface description of this class is as follows:

```
class DataReader
{
//
// inherited from class Entity
//
// StatusCondition_ptr
```

```

//      get_statuscondition
//      (void);
//      StatusMask
//      get_status_changes
//      (void);
//      ReturnCode_t
//      enable
//      (void);
//
//      abstract operations (implemented in the data type
//      specific DataReader)
//
//      ReturnCode_t
//      read
//      (<data>Seq& data_values,
//       SampleInfoSeq& info_seq,
//       Long max_samples,
//       SampleStateMask sample_states,
//       ViewStateMask view_states,
//       InstanceStateMask instance_states);
//      ReturnCode_t
//      take
//      (<data>Seq& data_values,
//       SampleInfoSeq& info_seq,
//       Long max_samples,
//       SampleStateMask sample_states,
//       ViewStateMask view_states,
//       InstanceStateMask instance_states);
//      ReturnCode_t
//      read_w_condition
//      (<data>Seq& data_values,
//       SampleInfoSeq& info_seq,
//       Long max_samples,
//       ReadCondition a_condition);
//      ReturnCode_t
//      take_w_condition
//      (<data>Seq& data_values,
//       SampleInfoSeq& info_seq,
//       Long max_samples,
//       ReadCondition a_condition);
//      ReturnCode_t
//      read_next_sample
//      (<data>& data_values,
//       SampleInfo sample_info);
//      ReturnCode_t
//      take_next_sample
//      (<data>& data_values,
//       SampleInfo sample_info);
//      ReturnCode_t
//      read_instance

```

```

//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq,
//      Long max_samples,
//      InstanceHandle_t a_handle,
//      SampleStateMask sample_states,
//      ViewStateMask view_states,
//      InstanceStateMask instance_states);
// ReturnCode_t
//      take_instance
//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq,
//      Long max_samples,
//      InstanceHandle_t a_handle,
//      SampleStateMask sample_states,
//      ViewStateMask view_states,
//      InstanceStateMask instance_states);
// ReturnCode_t
//      read_next_instance
//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq,
//      Long max_samples,
//      InstanceHandle_t a_handle,
//      SampleStateMask sample_states,
//      ViewStateMask view_states,
//      InstanceStateMask instance_states);
// ReturnCode_t
//      take_next_instance
//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq,
//      Long max_samples,
//      InstanceHandle_t a_handle,
//      SampleStateMask sample_states,
//      ViewStateMask view_states,
//      InstanceStateMask instance_states);
// ReturnCode_t
//      read_next_instance_w_condition
//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq,
//      Long max_samples,
//      InstanceHandle_t a_handle,
//      ReadCondition a_condition);
// ReturnCode_t
//      take_next_instance_w_condition
//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq,
//      Long max_samples,
//      InstanceHandle_t a_handle,
//      ReadCondition a_condition);
// ReturnCode_t
//      return_loan

```

```

//      (<data>Seq& data_values,
//      SampleInfoSeq& info_seq);
// ReturnCode_t
//      get_key_value
//      (<data>& key_holder,
//      InstanceHandle_t handle);
// InstanceHandle_t
//      lookup_instance
//      (const <data>& instance_data);
//
// implemented API operations
//
    ReadCondition_ptr
        create_readcondition
            (SampleStateMask sample_states,
             ViewStateMask view_states,
             InstanceStateMask instance_states);

    QueryCondition_ptr
        create_querycondition
            (SampleStateMask sample_states,
             ViewStateMask view_states,
             InstanceStateMask instance_states,
             const char* query_expression,
             const StringSeq& query_parameters);

    ReturnCode_t
        delete_readcondition
            (ReadCondition_ptr a_condition);

    ReturnCode_t
        delete_contained_entities
            (void);

    ReturnCode_t
        set_qos
            (const DataReaderQos& qos);

    ReturnCode_t
        get_qos
            (DataReaderQos& qos);

    ReturnCode_t
        set_listener
            (DataReaderListener_ptr a_listener,
             StatusMask mask);

    DataReaderListener_ptr
        get_listener
            (void);

```

```

    TopicDescription_ptr
        get_topicdescription
            (void);

    Subscriber_ptr
        get_subscriber
            (void);

    ReturnCode_t
        get_sample_rejected_status
            (SampleRejectedStatus& status);

    ReturnCode_t
        get_liveliness_changed_status
            (LivelinessChangedStatus& status);

    ReturnCode_t
        get_requested_deadline_missed_status
            (RequestedDeadlineMissedStatus& status);

    ReturnCode_t
        get_requested_incompatible_qos_status
            (RequestedIncompatibleQosStatus& status);

    ReturnCode_t
        get_subscription_matched_status
            (SubscriptionMatchedStatus& status);

    ReturnCode_t
        get_sample_lost_status
            (SampleLostStatus& status);

    ReturnCode_t
        wait_for_historical_data
            (const Duration_t& max_wait);

    ReturnCode_t
        get_matched_publications
            (InstanceHandleSeq& publication_handles);

    ReturnCode_t
        get_matched_publication_data
            (PublicationBuiltinTopicData& publication_data,
             InstanceHandle_t publication_handle);
};

```

The next paragraphs describe the usage of all `DataReader` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the

classes from which they are inherited. The abstract operations are listed but not fully described because they are not implemented in this specific class. The full description of these operations is located in the subclasses that contain the data type specific implementation of these operations.

3.5.2.2 create_querycondition

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
QueryCondition_ptr
create_querycondition
    (SampleStateMask sample_states,
     ViewStateMask view_states,
     InstanceStateMask instance_states,
     const char* query_expression,
     const StringSeq& query_parameters);
```

Description

This operation creates a new QueryCondition for the DataReader.

Parameters

- in SampleStateMask sample_states* - a mask, which selects only those samples with the desired sample states.
- in ViewStateMask view_states* - a mask, which selects only those samples with the desired view states.
- in InstanceStateMask instance_states* - a mask, which selects only those samples with the desired instance states.
- in const char* query_expression* - the query string, which must be a subset of the SQL query language.
- in const StringSeq& query_parameters* - a sequence of strings which are the parameter values used in the SQL query string (i.e., the “%n” tokens in the expression). The number of values in *query_parameters* must be equal or greater than the highest referenced %n token in the *query_expression* (e.g. if %1 and %8 are used as parameters in the *query_expression*, the *query_parameters* should at least contain $n+1 = 9$ values).

Return Value

QueryCondition_ptr - Result value is a pointer to the QueryCondition. When the operation fails, the NULL pointer is returned.

Detailed Description

This operation creates a new `QueryCondition` for the `DataReader`. The returned `QueryCondition` is attached (and belongs) to the `DataReader`. When the operation fails, the `NULL` pointer is returned. To delete the `QueryCondition` the operation `delete_readcondition` or `delete_contained_entities` must be used.

State Masks

The result of the `QueryCondition` also depends on the selection of samples determined by three masks:

- `sample_states` is the mask, which selects only those samples with the desired sample states `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both
- `view_states` is the mask, which selects only those samples with the desired view states `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE` or both
- `instance_states` is the mask, which selects only those samples with the desired instance states `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these.

SQL Expression

The SQL query string is set by `query_expression` which must be a subset of the SQL query language. In this query expression, parameters may be used, which must be set in the sequence of strings defined by the parameter `query_parameters`. A parameter is a string which can define an integer, float, string or enumeration. The number of values in `query_parameters` must be equal or greater than the highest referenced `%n` token in the `query_expression` (e.g. if `%1` and `%8` are used as parameters in the `query_expression`, the `query_parameters` should at least contain `n+1 = 9` values).

3.5.2.3 `create_readcondition`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReadCondition_ptr
    create_readcondition
        (SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
```


Description

This operation creates a new `ReadCondition` for the `DataReader`.

Parameters

in `SampleStateMask sample_states` - a mask, which selects only those samples with the desired sample states.

in `ViewStateMask view_states` - a mask, which selects only those samples with the desired view states.

in `InstanceStateMask instance_states` - a mask, which selects only those samples with the desired instance states.

Return Value

`ReadCondition_ptr` - Result value is a pointer to the `ReadCondition`. When the operation fails, the `NULL` pointer is returned.

Detailed Description

This operation creates a new `ReadCondition` for the `DataReader`. The returned `ReadCondition` is attached (and belongs) to the `DataReader`. When the operation fails, the `NULL` pointer is returned. To delete the `ReadCondition` the operation `delete_readcondition` or `delete_contained_entities` must be used.

State Masks

The result of the `ReadCondition` depends on the selection of samples determined by three masks:

- `sample_states` is the mask, which selects only those samples with the desired sample states `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both
- `view_states` is the mask, which selects only those samples with the desired view states `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE` or both
- `instance_states` is the mask, which selects only those samples with the desired instance states `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these.

3.5.2.4 `delete_contained_entities`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_contained_entities
        (void);
```

Description

This operation deletes all the Entity objects that were created by means of one of the “create_” operations on the DataReader.

Parameters

<none>

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation deletes all the Entity objects that were created by means of one of the “create_” operations on the DataReader. In other words, it deletes all QueryCondition and ReadCondition objects contained by the DataReader.



Note: The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted. In such cases, the operation does not roll-back any entity deletions performed prior to the detection of the problem.

Return Code

When the operation returns:

- *RETCODE_OK* - the contained Entity objects are deleted and the application may delete the DataReader
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - one or more of the contained entities are in a state where they cannot be deleted.

3.5.2.5 delete_readcondition

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    delete_readcondition
        (ReadCondition_ptr a_condition);
```

Description

This operation deletes a ReadCondition or QueryCondition which is attached to the DataReader.

Parameters

in ReadCondition_ptr a_condition - a pointer to the ReadCondition or QueryCondition which is to be deleted.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES or RETCODE_PRECONDITION_NOT_MET.

Detailed Description

This operation deletes a ReadCondition or QueryCondition which is attached to the DataReader. Since a QueryCondition is a specialized ReadCondition, the operation can also be used to delete a QueryCondition. A ReadCondition or QueryCondition cannot be deleted when it is not attached to this DataReader. When the operation is called on a ReadCondition or QueryCondition which was not attached to this DataReader, the operation returns RETCODE_PRECONDITION_NOT_MET.

Return Code

When the operation returns:

- *RETCODE_OK* - the ReadCondition or QueryCondition is deleted
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter *a_condition* is not a valid ReadCondition_ptr
- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted

- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_PRECONDITION_NOT_MET* - the operation is called on a different `DataReader`, as used when the `ReadCondition` or `QueryCondition` was created.

3.5.2.6 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
    ReturnCode_t
        enable
            (void);
```

3.5.2.7 get_key_value (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
    ReturnCode_t
        get_key_value
            (<data>& key_holder,
             InstanceHandle_t handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.8 get_listener

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
DataReaderListener_ptr
    get_listener
        (void);
```

Description

This operation allows access to a `DataReaderListener`.

Parameters

<none>

Return Value

datareaderlistener_ptr - result is a pointer to the `DataReaderListener` attached to the `DataReader`.

Detailed Description

This operation allows access to a `DataReaderListener` attached to the `DataReader`. When no `DataReaderListener` was attached to the `DataReader`, the `NULL` pointer is returned.

3.5.2.9 get_liveliness_changed_status**Scope**

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_liveliness_changed_status
        (LivelinessChangedStatus& status);
```

Description

This operation obtains the `LivelinessChangedStatus` struct of the `DataReader`.

Parameters

inout LivelinessChangedStatus& status - the contents of the `LivelinessChangedStatus` struct of the `DataReader` will be copied into the location specified by *status*.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This obtains returns the `LivelinessChangedStatus` struct of the `DataReader`. This struct contains the information whether the liveliness of one or more `DataWriter` objects that were writing instances read by the `DataReader` has changed. In other words, some `DataWriter` have become “alive” or “not alive”.

The `LivelinessChangedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `LivelinessChangedStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataReader` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.10 `get_matched_publication_data`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_matched_publication_data
        (PublicationBuiltinTopicData& publication_data,
         InstanceHandle_t publication_handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.11 `get_matched_publications`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_matched_publications
        (InstanceHandleSeq& publication_handles);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.12 `get_qos`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_qos
        (DataReaderQos& qos);
```

Description

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

Parameters

inout DataReaderQos& qos - a pointer to the destination DataReaderQos struct in which the QoSPolicy settings will be copied.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation allows access to the existing set of QoS policies of a DataReader on which this operation is used. This DataReaderQos is stored at the location pointed to by the qos parameter.

Return Code

When the operation returns:

- *RETCODE_OK* - the existing set of QoSPolicy values applied to this DataReader has successfully been copied into the specified DataReaderQos parameter.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.13 get_requested_deadline_missed_status

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_requested_deadline_missed_status
        (RequestedDeadlineMissedStatus& status);
```

Description

This operation obtains the `RequestedDeadlineMissedStatus` struct of the `DataReader`.

Parameters

inout `RequestedDeadlineMissedStatus& status` - the contents of the `RequestedDeadlineMissedStatus` struct of the `DataReader` will be copied into the location specified by `status`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `RequestedDeadlineMissedStatus` struct of the `DataReader`. This struct contains the information whether the deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected for a specific instance.

The `RequestedDeadlineMissedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `RequestedDeadlineMissedStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataReader` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.14 `get_requested_incompatible_qos_status`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_requested_incompatible_qos_status
        (RequestedIncompatibleQosStatus& status);
```


Description

This operation obtains the `RequestedIncompatibleQosStatus` struct of the `DataReader`.

Parameters

inout `RequestedIncompatibleQosStatus& status` - the contents of the `RequestedIncompatibleQosStatus` struct of the `DataReader` will be copied into the location specified by `status`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `RequestedIncompatibleQosStatus` struct of the `DataReader`. This struct contains the information whether a `QosPolicy` setting was incompatible with the offered `QosPolicy` setting.

The Request/Offering mechanism is applicable between the `DataWriter` and the `DataReader`. If the `QosPolicy` settings between `DataWriter` and `DataReader` are inconsistent, no communication between them is established. In addition the `DataWriter` will be informed via a `REQUESTED_INCOMPATIBLE_QOS` status change and the `DataReader` will be informed via an `OFFERED_INCOMPATIBLE_QOS` status change.

The `RequestedIncompatibleQosStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `RequestedIncompatibleQosStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataReader` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.15 `get_sample_lost_status`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_sample_lost_status
        (SampleLostStatus& status);
```

Description

This operation obtains the `SampleLostStatus` struct of the `DataReader`.

Parameters

inout SampleLostStatus& status - the contents of the `SampleLostStatus` struct of the `DataReader` will be copied into the location specified by *status*.

Note: This status is not yet implemented. It is scheduled for a future release. Until it is implemented all returned attribute values will be initialized to 0.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `SampleLostStatus` struct of the `DataReader`. This struct contains information whether samples have been lost. This only applies when the `ReliabilityQosPolicy` is set to `RELIABLE`. If the `ReliabilityQosPolicy` is set to `BEST_EFFORT` the Data Distribution Service will not report the loss of samples.

The `SampleLostStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `SampleLostStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataReader` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.16 `get_sample_rejected_status`

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_sample_rejected_status
        (SampleRejectedStatus& status);
```

Detailed Description

This operation obtains the `SampleRejectedStatus` struct of the `DataReader`.

Parameters

inout `SampleRejectedStatus& status` - the contents of the `SampleRejectedStatus` struct of the `DataReader` will be copied into the location specified by `status`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `SampleRejectedStatus` struct of the `DataReader`. This struct contains the information whether a received sample has been rejected.

The `SampleRejectedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the current `SampleRejectedStatus` of this `DataReader` has successfully been copied into the specified `status` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataReader` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.17 **get_status_changes (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusMask
    get_status_changes
        (void);
```

3.5.2.18 **get_statuscondition (inherited)**

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

3.5.2.19 **get_subscriber**

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
Subscriber_ptr
    get_subscriber
        (void);
```

Description

This operation returns the `Subscriber` to which the `DataReader` belongs.

Parameters

<none>

Return Value

Subscriber_ptr - Return value is a pointer to the `Subscriber` to which the `DataReader` belongs.

Detailed Description

This operation returns the Subscriber to which the DataReader belongs, thus the Subscriber that has created the DataReader. If the DataReader is already deleted, the NULL pointer is returned.

3.5.2.20 `get_subscription_matched_status`

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_subscription_matched_status
        (SubscriptionMatchedStatus& status);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.21 `get_topicdescription`

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
TopicDescription_ptr
    get_topicdescription
        (void);
```

Description

This operation returns the TopicDescription which is associated with the DataReader.

Parameters

<none>

Return Value

TopicDescription_ptr - Return value is a pointer to the TopicDescription which is associated with the DataReader.

Detailed Description

This operation returns the TopicDescription which is associated with the DataReader, thus the TopicDescription with which the DataReader is created. If the DataReader is already deleted, the NULL pointer is returned.

3.5.2.22 lookup_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceHandle_t
    lookup_instance
        (const <data>& instance_data);
```

3.5.2.23 read (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    read
        (<data>& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
```

3.5.2.24 read_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    read_instance
        (<data>& data_values,
         SampleInfoSeq& info_seq,
```

```

Long max_samples,
InstanceHandle_t a_handle,
SampleStateMask sample_states,
ViewStateMask view_states,
InstanceStateMask instance_states);

```

3.5.2.25 read_next_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
read_next_instance
(<data>& data_values,
 SampleInfoSeq& info_seq,
 Long max_samples,
 InstanceHandle_t a_handle,
 SampleStateMask sample_states,
 ViewStateMask view_states,
 InstanceStateMask instance_states);

```

3.5.2.26 read_next_instance_w_condition (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
read_next_instance_w_condition
(<data>& data_values,
 SampleInfoSeq& info_seq,
 Long max_samples,
 InstanceHandle_t a_handle,
 ReadCondition a_condition);

```

3.5.2.27 read_next_sample (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
      ReturnCode_t
      read_next_sample
      (<data>& data_value,
       SampleInfo sample_info);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.28 read_w_condition (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
      ReturnCode_t
      read_w_condition
      (<data>& data_values,
       SampleInfoSeq& info_seq,
       Long max_samples,
       ReadCondition a_condition);
```

3.5.2.29 return_loan (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
      ReturnCode_t
      return_loan
      (<data>& data_values,
       SampleInfoSeq& info_seq);
```


3.5.2.30 **set_listener**

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_listener
        (DataReaderListener_ptr a_listener,
         StatusMask mask);
```

Description

This operation attaches a DataReaderListener to the DataReader.

Parameters

in DataReaderListener_ptr a_listener - a pointer to the DataReaderListener instance, which will be attached to the DataReader.

in StatusMask mask - a bit mask in which each bit enables the invocation of the DataReaderListener for a certain status.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation attaches a DataReaderListener to the DataReader. Only one DataReaderListener can be attached to each DataReader. If a DataReaderListener was already attached, the operation will replace it with the new one. When *a_listener* is the NULL pointer, it represents a listener that is treated as a NOOP¹ for all statuses activated in the bit mask.

Communication Status

For each communication status, the StatusChangedFlag flag is initially set to FALSE. It becomes TRUE whenever that communication status changes. For each communication status activated in the mask, the associated DataReaderListener operation 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

1. Short for **No-Operation**, an instruction that does nothing.

`get_<status_name>_status` from inside the listener it will see the status already reset. An exception to this rule is the `NULL` listener, which does not reset the communication statuses for which it is invoked.

The following statuses are applicable to the `DataReaderListener`:

- `REQUESTED_DEADLINE_MISSED_STATUS`
- `REQUESTED_INCOMPATIBLE_QOS_STATUS`
- `SAMPLE_LOST_STATUS`
- `SAMPLE_REJECTED_STATUS`
- `DATA_AVAILABLE_STATUS`
- `LIVELINESS_CHANGED_STATUS`
- `SUBSCRIPTION_MATCHED_STATUS`.

Status bits are declared as a constant and can be used by the application in an OR operation to create a tailored mask. The special constant `DDS_STATUS_MASK_NONE` can be used to indicate that the created entity should not respond to any of its available statuses. The special constant `STATUS_MASK_ANY_V1_2` can be used to select all statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

Status Propagation

In case a communication status is not activated in the `mask`, the `SubscriberListener` of the `DataReaderListener` is invoked (if attached and activated for the status that occurred). This allows the application to set a default behaviour in the `SubscriberListener` of the containing `Subscriber` and a `DataReader` specific behaviour when needed. In case the communication status is not activated in the `mask` of the `SubscriberListener` as well, the communication status will be propagated to the `DomainParticipantListener` of the containing `DomainParticipant`. In case the `DomainParticipantListener` is also not attached or the communication status is not activated in its `mask`, the application is not notified of the change.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` are “Read Communication Statuses” and are an exception to all other plain communication statuses: they have no corresponding status structure that can be obtained with a `get_<status_name>_status` operation and they are mutually exclusive. When new information becomes available to a `DataReader`, the Data Distribution Service will first look in an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look in an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

Return Code

When the operation returns:

- *RETCODE_OK* - the *DataReaderListener* is attached
- *RETCODE_ERROR* - an internal error has occurred
- *RETCODE_ALREADY_DELETED* - the *DataReader* has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

3.5.2.31 **set_qos**

Scope

`DDS::DataReader`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    set_qos
        (const DataReaderQos& qos);
```

Description

This operation replaces the existing set of *QosPolicy* settings for a *DataReader*.

Parameters

in const DataReaderQos& qos - qos contains the new set of *QosPolicy* settings for the *DataReader*.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_UNSUPPORTED*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES*, *RETCODE_IMMUTABLE_POLICY* or *RETCODE_INCONSISTENT_POLICY*.

Detailed Description

This operation replaces the existing set of *QosPolicy* settings for a *DataReader*. The parameter *qos* contains the *QosPolicy* settings which is checked for self-consistency and mutability. When the application tries to change a *QosPolicy* setting for an enabled *DataReader*, which can only be set before the *DataReader* is enabled, the operation will fail and a *RETCODE_IMMUTABLE_POLICY* is returned. In other words, the application must provide the presently set *QosPolicy* settings in case of the immutable *QosPolicy* settings. Only the mutable *QosPolicy*

settings can be changed. When qos contains conflicting QosPolicy settings (not self-consistent), the operation will fail and a RETCODE_INCONSISTENT_POLICY is returned.

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 RETCODE_OK).

Return Code

When the operation returns:

- *RETCODE_OK* - the new DataReaderQos is set
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter qos is not a valid DataReaderQos. It contains a QosPolicy setting with an invalid Duration_t value or an enum value that is outside its legal boundaries
- *RETCODE_UNSUPPORTED* - one or more of the selected QosPolicy values are currently not supported by OpenSplice.
- *RETCODE_ALREADY_DELETED* - the DataReader has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_IMMUTABLE_POLICY* - the parameter qos contains an immutable QosPolicy setting with a different value than set during enabling of the DataReader
- *RETCODE_INCONSISTENT_POLICY* - the parameter qos contains conflicting QosPolicy settings, e.g. a history depth that is higher than the specified resource limits.

3.5.2.32 take (abstract)

This abstract operation is defined as a generic operation, which is implemented by the <type>DataReader class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type Foo derived FooDataReader class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
take
    (<data>& data_values,
     SampleInfoSeq& info_seq,
     Long max_samples,
```

```

SampleStateMask sample_states,
ViewStateMask view_states,
InstanceStateMask instance_states)

```

3.5.2.33 take_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
take_instance
(
<data>& data_values,
SampleInfoSeq& info_seq,
Long max_samples,
InstanceHandle_t a_handle,
SampleStateMask sample_states,
ViewStateMask view_states,
InstanceStateMask instance_states);

```

3.5.2.34 take_next_instance (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```

#include <ccpp_dds_dcps.h>
ReturnCode_t
take_next_instance
(
<data>& data_values,
SampleInfoSeq& info_seq,
Long max_samples,
InstanceHandle_t a_handle,
SampleStateMask sample_states,
ViewStateMask view_states,
InstanceStateMask instance_states);

```

3.5.2.35 take_next_instance_w_condition (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    take_next_instance_w_condition
        (<data>& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         ReadCondition a_condition);
```

3.5.2.36 take_next_sample (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    take_next_sample
        (<data>& data_value,
         SampleInfo sample_info);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.37 take_w_condition (abstract)

This abstract operation is defined as a generic operation, which is implemented by the `<type>DataReader` class. Therefore, to use this operation, the data type specific implementation of this operation in its respective derived class must be used. For further explanation see the description for the fictional data type `Foo` derived `FooDataReader` class.

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    take_w_condition
        (<data>& data_values,
```

```
SampleInfoSeq& info_seq,
Long max_samples,
ReadCondition a_condition);
```

3.5.2.38 wait_for_historical_data

Scope

DDS::DataReader

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
wait_for_historical_data
(const Duration_t& max_wait);
```

Description

This operation will block the application thread until all “historical” data is received.

Parameters

in const Duration_t& max_wait - the maximum duration to block for the `wait_for_historical_data`, after which the application thread is unblocked. The special constant `DURATION_INFINITE` can be used when the maximum waiting time does not need to be bounded.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED` or `RETCODE_TIMEOUT`.

Detailed Description

This operation behaves differently for `DataReader` objects which have a `non-VOLATILE_DURABILITY_QOS` `DurabilityQosPolicy` and for `DataReader` objects which have a `VOLATILE_DURABILITY_QOS` `DurabilityQosPolicy`.

As soon as an application enables a `non-VOLATILE_DURABILITY_QOS` `DataReader` it will start receiving both “historical” data, i.e. the data that was written prior to the time the `DataReader` joined the domain, as well as any new data written by the `DataWriter` objects. There are situations where the application logic may require the application to wait until all “historical” data is received. This is the purpose of the `wait_for_historical_data` operation.

As soon as an application enables a `VOLATILE_DURABILITY_QOS` `DataReader` it will not start receiving “historical” data but only new data written by the `DataWriter` objects. By calling `wait_for_historical_data` the `DataReader` explicitly requests the Data Distribution Service to start receiving also the “historical” data and to wait until either all “historical” data is received, or the duration specified by the `max_wait` parameter has elapsed, whichever happens first.

Thread Blocking

The operation `wait_for_historical_data` blocks the calling thread until either all “historical” data is received, or the duration specified by the `max_wait` parameter elapses, whichever happens first. A return value of `RETCODE_OK` indicates that all the “historical” data was received a return value of `RETCODE_TIMEOUT` indicates that `max_wait` elapsed before all the data was received.

Return Code

When the operation returns:

- `RETCODE_OK` - the “historical” data is received
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `DataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `DataReader` is not enabled.
- `RETCODE_TIMEOUT` - not all data is received before `max_wait` elapsed.

3.5.2.39 Class `FooDataReader`

The pre-processor generates from IDL type descriptions the application `<type>DataReader` classes. For each application data type that is used as `Topic` data type, a typed class `<type>DataReader` is derived from the `DataReader` class. In this paragraph, the class `FooDataReader` in the namespace `SPACE` describes the operations of these derived `<type>DataReader` classes as an example for the fictional application type `Foo` (defined in the module `SPACE`).

For instance, for an application, the definitions are located in the `Space.idl` file. The pre-processor will generate a `ccpp_Space.h` include file.



General note: The name `ccpp_Space.h` is derived from the IDL file `Space.idl`, that defines `Foo`, for all relevant `FooDataWriter` operations.

State Masks

A `FooDataReader` refers to exactly one `TopicDescription` (either a `Topic`, a `ContentFilteredTopic` or a `MultiTopic`) that identifies the data to be read. Therefore it refers to exactly one data type. The `Topic` must exist prior to the `FooDataReader` creation. The `FooDataReader` may give access to several instances of the data type, which are distinguished from each other by their `key`. The `FooDataReader` is attached to exactly one `Subscriber` which acts as a factory for it.

The interface description of this class is as follows:

```
class FooDataReader
{
//
// inherited from class Entity
//
// StatusCondition_ptr
//   get_statuscondition
//   (void);
// StatusMask
//   get_status_changes
//   (void);
// ReturnCode_t
//   enable
//   (void);
//
// inherited from class DataReader
//
// ReadCondition_ptr
//   create_readcondition
//   (SampleStateMask sample_states,
//    ViewStateMask view_states,
//    InstanceStateMask instance_states);

// QueryCondition_ptr
//   create_querycondition
//   (SampleStateMask sample_states,
//    ViewStateMask view_states,
//    InstanceStateMask instance_states,
//    const char* query_expression,
//    const StringSeq& query_parameters);

// ReturnCode_t
//   delete_readcondition
//   (ReadCondition_ptr a_condition);

// ReturnCode_t
//   delete_contained_entities
//   (void);
```

```

// ReturnCode_t
//     set_qos
//         (const DataReaderQos& qos);

// ReturnCode_t
//     get_qos
//         (DataReaderQos& qos);

// ReturnCode_t
//     set_listener
//         (DataReaderListener_ptr a_listener,
//          StatusMask mask);

// DataReaderListener_ptr
//     get_listener
//         (void);

// TopicDescription_ptr
//     get_topicdescription
//         (void);

// Subscriber_ptr
//     get_subscriber
//         (void);

// ReturnCode_t
//     get_sample_rejected_status
//         (SampleRejectedStatus& status);

// ReturnCode_t
//     get_liveliness_changed_status
//         (LivelinessChangedStatus& status);

// ReturnCode_t
//     get_requested_deadline_missed_status
//         (RequestedDeadlineMissedStatus& status);

// ReturnCode_t
//     get_requested_incompatible_qos_status
//         (RequestedIncompatibleQosStatus& status);

// ReturnCode_t
//     get_subscription_matched_status
//         (SubscriptionMatchedStatus& status);

// ReturnCode_t
//     get_sample_lost_status
//         (SampleLostStatus& status);

```

```

// ReturnCode_t
//     wait_for_historical_data
//         (const Duration_t& max_wait);

// ReturnCode_t
//     get_matched_publications
//         (InstanceHandleSeq& publication_handles);

// ReturnCode_t
//     get_matched_publication_data
//         (PublicationBuiltinTopicData& publication_data,
//          InstanceHandle_t publication_handle);
//
// implemented API operations
//
ReturnCode_t
    read
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
ReturnCode_t
    take
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
ReturnCode_t
    read_w_condition
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         ReadCondition_ptr a_condition);
ReturnCode_t
    take_w_condition
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         ReadCondition_ptr a_condition);
ReturnCode_t
    data_value
        (Foo& received_data,
         SampleInfo sample_info);
ReturnCode_t
    take_next_sample
        (Foo& data_value,

```

```

        SampleInfo sample_info);
ReturnCode_t
    read_instance
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
ReturnCode_t
    take_instance
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
ReturnCode_t
    read_next_instance
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
ReturnCode_t
    take_next_instance
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
ReturnCode_t
    read_next_instance_w_condition
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         ReadCondition_ptr a_condition);
ReturnCode_t
    take_next_instance_w_condition
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,

```

```

        ReadCondition_ptr a_condition);
ReturnCode_t
    return_loan
        (FooSeq& data_values,
         SampleInfoSeq& info_seq);
ReturnCode_t
    get_key_value
        (Foo& key_holder,
         InstanceHandle_t handle);
InstanceHandle_t
    lookup_instance
        (const Foo& instance_data);
};

```

The next paragraphs describe the usage of all `FooDataReader` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.2.40 `create_querycondition` (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```

#include <ccpp_Space.h>
QueryCondition_ptr
    create_querycondition
        (SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states,
         const char* query_expression,
         const StringSeq& query_parameters);

```

3.5.2.41 `create_readcondition` (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```

#include <ccpp_Space.h>
ReadCondition_ptr
    create_readcondition
        (SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);

```

3.5.2.42 delete_contained_entities (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    delete_contained_entities
        (void);
```

3.5.2.43 delete_readcondition (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    delete_readcondition
        (ReadCondition_ptr a_condition);
```

3.5.2.44 enable (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    enable
        (void);
```

3.5.2.45 get_key_value

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_key_value
        (Foo& key_holder,
         InstanceHandle_t handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.46 get_listener (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
DataReaderListener_ptr
    get_listener
        (void);
```

3.5.2.47 get_liveliness_changed_status (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_liveliness_changed_status
        (LivelinessChangedStatus& status);
```

3.5.2.48 get_matched_publication_data (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_matched_publication_data
        (PublicationBuiltinTopicData& publication_data,
         InstanceHandle_t publication_handle);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.49 get_matched_publications (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_matched_publications
        (InstanceHandleSeq& publication_handles);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.50 **get_qos (inherited)**

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_qos
        (DataReaderQos& qos);
```

3.5.2.51 **get_requested_deadline_missed_status (inherited)**

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_requested_deadline_missed_status
        (RequestedDeadlineMissedStatus& status);
```

3.5.2.52 **get_requested_incompatible_qos_status (inherited)**

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_requested_incompatible_qos_status
        (RequestedIncompatibleQosStatus& status);
```

3.5.2.53 **get_sample_lost_status (inherited)**

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_sample_lost_status
        (SampleLostStatus& status);
```

3.5.2.54 **get_sample_rejected_status (inherited)**

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_sample_rejected_status
        (SampleRejectedStatus& status);
```

3.5.2.55 get_status_changes (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
StatusMask
    get_status_changes
        (void);
```

3.5.2.56 get_statuscondition (inherited)

This operation is inherited and therefore not described here. See the class `Entity` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
StatusCondition_ptr
    get_statuscondition
        (void);
```

3.5.2.57 get_subscriber (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
Subscriber_ptr
    get_subscriber
        (void);
```

3.5.2.58 get_subscription_matched_status (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    get_subscription_matched_status
```

```
(SubscriptionMatchedStatus& status);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.59 `get_topicdescription` (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
TopicDescription_ptr
    get_topicdescription
        (void);
```

3.5.2.60 `lookup_instance`

Scope

```
SPACE::FooDataReader
```

Synopsis

```
#include <ccpp_Space.h>
InstanceHandle_t
    lookup_instance
        (const Foo& instance_data);
```

Description

This operation returns the value of the instance handle which corresponds to the `instance_data`.

Parameters

in const Foo& instance_data - the instance for which the corresponding instance handle needs to be looked up.

Return Value

InstanceHandle_t - Result value is the instance handle which corresponds to the `instance_data`.

Detailed Description

This operation returns the value of the instance handle which corresponds to the `instance_data`. The instance handle can be used in read operations that operate on a specific instance. Note that `DataReader` instance handles are local, and are not interchangeable with `DataWriter` instance handles nor with instance handles of an other `DataReader`. If the `DataReader` is already deleted, the handle value `HANDLE_NIL` is returned.

3.5.2.61 read

Scope

SPACE::FooDataReader

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
read
(
    FooSeq& data_values,
    SampleInfoSeq& info_seq,
    Long max_samples,
    SampleStateMask sample_states,
    ViewStateMask view_states,
    InstanceStateMask instance_states);
```

Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned `SampleInfo` structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in SampleStateMask sample_states - a mask, which selects only those samples with the desired sample states.

in ViewStateMask view_states - a mask, which selects only those samples with the desired view states.

in InstanceStateMask instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`. The data is returned by the parameters `data_values` and `info_seq`. The number of samples that is returned is limited by the parameter `max_samples`. This operation is part of the specialized class which is generated for the particular application data type (in this case type `Foo`) that is being read. If the `FooDataReader` has no samples that meet the constraints, the return value is `RETCODE_NO_DATA`.

State Masks

The read operation depends on a selection of the samples by using three masks:

- `sample_states` is the mask, which selects only those samples with the desired sample states `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both
- `view_states` is the mask, which selects only those samples with the desired view states `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE` or both
- `instance_states` is the mask, which selects only those samples with the desired instance states `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these.

Destination Order

In any case, the relative order between the samples of one instance is consistent with the `DestinationOrderQosPolicy` of the Subscriber.

- When the `DestinationOrderQosPolicy` kind is `BY_RECEPTION_timestamp_DESTINATIONORDER_QOS`, the samples belonging to the same instances will appear in the relative order in which they were received (FIFO)
- When the `DestinationOrderQosPolicy` kind is `BY_SOURCE_timestamp_DESTINATIONORDER_QOS`, the samples belonging to the same instances will appear in the relative order implied by the `source_timestamp`.

Data Sample

In addition to the sample sequence (`data_values`), the operation also returns a sequence of `SampleInfo` structures with the parameter `info_seq`. The `info_seq` structures and `data_values` also determine the behaviour of this operation.

Resource Control

The initial (input) properties of the `data_values` and `info_seq` sequences determine the precise behaviour of the `read` operation. The sequences are modelled as having three properties: the current-length (`length`), the maximum length (`maximum`), and whether or not the sequence container owns the memory of the elements within (`release`).

The initial (input) values of the `length`, `maximum`, and `release` properties for the `data_values` and `info_seq` sequences govern the behaviour of the `read` operation as specified by the following rules:

- The values of `length`, `maximum`, and `release` for the two sequences must be identical. Otherwise `read` returns `RETCODE_PRECONDITION_NOT_MET`
- On successful output, the values of `length`, `maximum`, and `release` are the same for both sequences
- If the input `maximum == 0`, the `received_data` and `info_seq` sequences are filled with elements that are “loaned” by the `FooDataReader`. On output, `release` is `FALSE`, `length` is set to the number of values returned, and `maximum` is set to a value verifying `maximum >= length`. In this case the application will need to “return the loan” to the Data Distribution Service using the `return_loan` operation
- If the input `maximum > 0` and the input `release == FALSE`, the `read` operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`. This avoids the potential hard-to-detect memory leaks caused by an application forgetting to “return the loan”
- If input `maximum > 0` and the input `release == TRUE`, the `read` operation will copy the `Foo` samples and `info_seq` values into the elements already inside the sequences. On output, `release` is `TRUE`, `length` is set to the number of values copied, and `maximum` will remain unchanged. The application can control where the copy is placed and the application does not need to “return the loan”. The number of samples copied depends on the relative values of `maximum` and `max_samples`:
 - If `max_samples == LENGTH_UNLIMITED`, at most `maximum` values are copied. The use of this variant lets the application limit the number of samples returned to what the sequence can accommodate
 - If `max_samples <= maximum`, at most `max_samples` values are copied. The use of this variant lets the application limit the number of samples returned to fewer than what the sequence can accommodate

- If `max_samples > maximum`, the `read` operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`. This avoids the potential confusion where the application expects to be able to access up to `max_samples`, but that number can never be returned, even if they are available in the `FoodDataReader`, because the output sequence cannot accommodate them.

Buffer Loan

As described above, upon return the `data_values` and `info_seq` sequences may contain elements “loaned” from the Data Distribution Service. If this is the case, the application will need to use the `return_loan` operation to return the “loan” once it is no longer using the data in the sequence. Upon return from `return_loan`, the sequence has `maximum==0` and `release==FALSE`.

The application can determine whether it is necessary to “return the loan” or not, based on the state of the sequences, when the `read` operation was called, or by accessing the “release” property. However, in many cases it may be simpler to always call `return_loan`, as this operation is harmless (i.e. leaves all elements unchanged) if the sequence does not have a loan.

To avoid potential memory leaks, it is not allowed to change the length of the `data_values` and `info_seq` structures for which `release==FALSE`. Furthermore, deleting a sequence for which `release==FALSE` is considered to be an error except when the sequence is empty.

Data Sequence

On output, the sequence of data values and the sequence of `SampleInfo` structures are of the same length and are in an one-to-one correspondence. Each `SampleInfo` structures provides information, such as the `source_timestamp`, the `sample_state`, `view_state`, and `instance_state`, etc., about the matching sample.

Some elements in the returned sequence may not have valid data: the `valid_data` field in the `SampleInfo` indicates whether the corresponding data value contains any meaningful data. If not, the data value is just a ‘dummy’ sample for which only the keyfields have been assigned. It is used to accompany the `SampleInfo` that communicates a change in the `instance_state` of an instance for which there is no ‘real’ sample available.

For example, when an application always ‘takes’ all available samples of a particular instance, there is no sample available to report the disposal of that instance. In such a case the `DataReader` will insert a dummy sample into the `data_values` sequence to accompany the `SampleInfo` element in the `info_seq` sequence that communicates the disposal of the instance.

The act of reading a sample sets its `sample_state` to `READ_SAMPLE_STATE`. If the sample belongs to the most recent generation of the instance, it also sets the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It does not affect the `instance_state` of the instance.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `max_samples > maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum > 0` and the `release == FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.62 `read_instance`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
      ReturnCode_t
      read_instance
      (FooSeq& data_values,
       SampleInfoSeq& info_seq,
       Long max_samples,
       InstanceHandle_t a_handle,
       SampleStateMask sample_states,
       ViewStateMask view_states,
       InstanceStateMask instance_states);
```

Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned `SampleInfo` structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in InstanceHandle_t a_handle - the single instance, the samples belong to.

in SampleStateMask sample_states - a mask, which selects only those samples with the desired sample states.

in ViewStateMask view_states - a mask, which selects only those samples with the desired view states.

in InstanceStateMask instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`. The behaviour is identical to `read` except for that all samples returned belong to the single specified instance whose handle is *a_handle*. Upon successful return, the data collection will contain samples all belonging to the same instance. The data is returned by the parameters *data_values* and *info_seq*. The corresponding `SampleInfo.instance_handle` in *info_seq* will have the value of *a_handle*. The `DataReader` will check that each sample belongs to the specified instance (indicated by *a_handle*) otherwise it will not place the sample in the returned collection.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available
- `RETCODE_ERROR` - an internal error has occurred.

- `RETCODE_BAD_PARAMETER` - the parameter `a_handle` is not a valid handle
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `max_samples > maximum` and `max_samples` is not `LENGTH_UNLIMITED`.
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical.
 - the `maximum > 0` and the `release == FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.63 `read_next_instance`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
    read_next_instance
    (FooSeq& data_values,
     SampleInfoSeq& info_seq,
     Long max_samples,
     InstanceHandle_t a_handle,
     SampleStateMask sample_states,
     ViewStateMask view_states,
     InstanceStateMask instance_states);
```

Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader`.

Parameters

inout `FooSeq& data_values` - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout `SampleInfoSeq& info_seq` - the returned `SampleInfo` structure sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in `long max_samples` - the maximum number of samples that is returned.

in InstanceHandle_t a_handle - the current single instance, the returned samples belong to the next single instance.

in SampleStateMask sample_states - a mask, which selects only those samples with the desired sample states.

in ViewStateMask view_states - a mask, which selects only those samples with the desired view states.

in InstanceStateMask instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_NO_DATA.

Detailed Description

This operation reads a sequence of Foo samples of a single instance from the FooDataReader. The behaviour is similar to *read_instance* (all samples returned belong to a single instance) except that the actual instance is not directly specified. Rather the samples will all belong to the ‘next’ instance with *instance_handle* ‘greater’ (according to some internal-defined order) than *a_handle*, that has available samples. The data is returned by the parameters *data_values* and *info_seq*. The corresponding *SampleInfo.instance_handle* in *info_seq* will have the value of the next instance with respect to *a_handle*.

Instance Order

The internal-defined order is not important and is implementation specific. The important thing is that, according to the Data Distribution Service, all instances are ordered relative to each other. This ordering is between the instances, that is, it does not depend on the actual samples received. For the purposes of this explanation it is ‘as if’ each instance handle was represented as a unique integer.

The behaviour of *read_next_instance* is ‘as if’ the DataReader invoked *read_instance* passing the smallest *instance_handle* among all the ones that:

- are greater than *a_handle*
- have available samples (i.e. samples that meet the constraints imposed by the specified states).

- The special value `HANDLE_NIL` is guaranteed to be 'less than' any valid `instance_handle`. So the use of the parameter value `a_handle==HANDLE_NIL` will return the samples for the instance which has the smallest `instance_handle` among all the instances that contains available samples.

Typical Use

The operation `read_next_instance` is intended to be used in an application-driven iteration where the application starts by passing `a_handle==HANDLE_NIL`, examines the samples returned, and then uses the `instance_handle` returned in the `SampleInfo` as the value of `a_handle` argument to the next call to `read_next_instance`. The iteration continues until `read_next_instance` returns the return value `RETCODE_NO_DATA`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_handle` is not a valid handle
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `max_samples>maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum>0` and the `release==FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.64 `read_next_instance_w_condition`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
    read_next_instance_w_condition
```

```
(FooSeq& data_values,
 SampleInfoSeq& info_seq,
 Long max_samples,
 InstanceHandle_t a_handle,
 ReadCondition_ptr a_condition);
```

Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition`.

Parameters

inout FooSeq& data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned `SampleInfo` structure sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in InstanceHandle_t a_handle - the current single instance, the returned samples belong to the next single instance.

in ReadCondition_ptr a_condition - a pointer to a `ReadCondition` or `QueryCondition` which filters the data before it is returned by the read operation.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition`. The behaviour is identical to `read_next_instance` except for that the samples are filtered by a `ReadCondition` or `QueryCondition`. When using a `ReadCondition`, the result is the same as the `read_next_instance` operation with the same state parameters filled in as for the `create_readcondition`. In this way, the application can avoid repeating the same parameters, specified when creating the `ReadCondition`. When using a `QueryCondition`, a content based

filtering can be done. When either using a `ReadCondition` or `QueryCondition`, the condition must be created by this `FooDataReader`. Otherwise the operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_condition` is not a valid `ReadCondition_ptr` or `a_handle` is not a valid handle.
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `max_samples > maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum > 0` and the `release == FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.65 `read_next_sample`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
    read_next_sample
    (Foo& data_value,
     SampleInfo sample_info);
```

Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.66 `read_w_condition`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
read_w_condition
(FooSeq& data_values,
 SampleInfoSeq& info_seq,
 Long max_samples,
 ReadCondition_ptr a_condition);
```

Description

This operation reads a sequence of Foo samples from the FooDataReader, filtered by a ReadCondition or QueryCondition.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned SampleInfo structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in ReadCondition_ptr a_condition - a pointer to a ReadCondition or QueryCondition which filters the data before it is returned by the read operation.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_NO_DATA.

Detailed Description

This operation reads a sequence of Foo samples from the FooDataReader, filtered by a ReadCondition or QueryCondition. The condition reference from both *create_readcondition* or *create_querycondition* may be used. The behaviour is identical to *read* except for that the samples are filtered by a ReadCondition or QueryCondition. When using a ReadCondition, the result is the same as the *read* operation with the same state parameters filled in as for the *create_readcondition*. In this way, the application can avoid repeating the same parameters, specified when creating the ReadCondition. When using a QueryCondition, a content based filtering can be done. When either using a

ReadCondition or QueryCondition, the condition must be created by this FooDataReader. Otherwise the operation will fail and returns RETCODE_PRECONDITION_NOT_MET.

Return Code

When the operation returns:

- *RETCODE_OK* - a sequence of data values is available
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the parameter a_condition is not a valid ReadCondition_ptr
- *RETCODE_ALREADY_DELETED* - the FooDataReader has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the FooDataReader is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - one of the following is true:
 - the ReadCondition or QueryCondition is not attached to this FooDataReader
 - the max_samples>maximum and max_samples is not LENGTH_UNLIMITED
 - one or more values of length, maximum, and release for the two sequences are not identical
 - the maximum>0 and the release==FALSE.
- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.67 return_loan

Scope

SPACE::FooDataReader

Synopsis

```
#include <ccpp_Space.h>
      ReturnCode_t
      return_loan
      (FooSeq& data_values,
       SampleInfoSeq& info_seq);
```

Description

This operation indicates to the DataReader that the application is done accessing the sequence of data_values and info_seq.

Parameters

inout FooSeq& data_values - the sample data sequence which was loaned from the `DataReader`.

inout SampleInfoSeq& info_seq - the `SampleInfo` structure sequence which was loaned from the `DataReader`.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED` or `RETCODE_PRECONDITION_NOT_MET`.

Detailed Description

This operation indicates to the `DataReader` that the application is done accessing the sequence of `data_values` and `info_seq` obtained by some earlier invocation of the operation `read` or `take` (or any of the similar operations) on the `DataReader`.

The `data_values` and `info_seq` must belong to a single related pair that is, they should correspond to a pair returned from a single call to the operation `read` or `take`. The `data_values` and `info_seq` must also have been obtained from the same `DataReader` to which they are returned. If either of these conditions is not met the operation will fail and returns `RETCODE_PRECONDITION_NOT_MET`.

Buffer Loan

The operation `return_loan` allows implementations of the `read` and `take` operations to “loan” buffers from the Data Distribution Service to the application and in this manner provide “zero-copy” access to the data. During the loan, the Data Distribution Service will guarantee that the `data_values` and `info_seq` are not modified.

It is not necessary for an application to return the loans immediately after calling the operation `read` or `take`. However, as these buffers correspond to internal resources inside the `DataReader`, the application should not retain them indefinitely.

Calling return_loan

The use of the `return_loan` operation is only necessary if the call to the operation `read` or `take` “loaned” buffers to the application. This only occurs if the `data_values` and `info_seq` sequences had `maximum=0` at the time the operation `read` or `take` was called. The application may also examine the ‘release’

property of the collection to determine where there is an outstanding loan. However, calling the operation `return_loan` on a pair of sequences that does not have a loan is safe and has no side effects.

If the pair of sequences had a loan, upon return from the operation `return_loan` the pair of sequences has `maximum=0`.

Return Code

When the operation returns:

- `RETCODE_OK` - the `DataReader` is informed that the sequences will not be used any more
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `data_values` and `info_seq` do not belong to a single related pair
 - the `data_values` and `info_seq` were not obtained from this `FooDataReader`.

3.5.2.68 `set_listener` (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
set_listener
(DataReaderListener_ptr a_listener,
 StatusMask mask);
```

3.5.2.69 `set_qos` (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
set_qos
(const DataReaderQos& qos);
```

3.5.2.70 take

Scope

SPACE::FooDataReader

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
take
    (FooSeq& data_values,
     SampleInfoSeq& info_seq,
     Long max_samples,
     SampleStateMask sample_states,
     ViewStateMask view_states,
     InstanceStateMask instance_states);
```

Description

This operation reads a sequence of Foo samples from the FooDataReader and by doing so, removes the data from the FooDataReader.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned SampleInfo structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in SampleStateMask sample_states - a mask, which selects only those samples with the desired sample states.

in ViewStateMask view_states - a mask, which selects only those samples with the desired view states.

in InstanceStateMask instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_NO_DATA.

Detailed Description

This operation reads a sequence of `Foo` samples from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available and removed from the `FooDataReader`
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `max_samples > maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum > 0` and the `release == FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.71 `take_instance`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
    take_instance
        (FooSeq& data_values,
         SampleInfoSeq& info_seq,
         Long max_samples,
         InstanceHandle_t a_handle,
         SampleStateMask sample_states,
         ViewStateMask view_states,
         InstanceStateMask instance_states);
```

Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned `SampleInfo` structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in InstanceHandle_t a_handle - the single instance, the samples belong to.

in SampleStateMask sample_states - a mask, which selects only those samples with the desired sample states.

in ViewStateMask view_states - a mask, which selects only those samples with the desired view states.

in InstanceStateMask instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_instance` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available and removed from the `FooDataReader`
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_handle` is not a valid handle

- *RETCODE_ALREADY_DELETED* - the `FooDataReader` has already been deleted
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.
- *RETCODE_NOT_ENABLED* - the `FooDataReader` is not enabled.
- *RETCODE_PRECONDITION_NOT_MET* - one of the following is true:
 - the `max_samples > maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum > 0` and the `release == FALSE`.
- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.72 `take_next_instance`

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
take_next_instance
(FooSeq& data_values,
 SampleInfoSeq& info_seq,
 Long max_samples,
 InstanceHandle_t a_handle,
 SampleStateMask sample_states,
 ViewStateMask view_states,
 InstanceStateMask instance_states);
```

Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`.

Parameters

inout FooSeq& data_values - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned `SampleInfo` structure sequence. `info_seq` is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in InstanceHandle_t a_handle - the current single instance, the returned samples belong to the next single instance.

in SampleStateMask sample_states - a mask, which selects only those samples with the desired sample states.

in ViewStateMask view_states - a mask, which selects only those samples with the desired view states.

in InstanceStateMask instance_states - a mask, which selects only those samples with the desired instance states.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_BAD_PARAMETER`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES`, `RETCODE_NOT_ENABLED`, `RETCODE_PRECONDITION_NOT_MET` or `RETCODE_NO_DATA`.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_next_instance` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available and removed from the `FooDataReader`
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_handle` is not a valid handle
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `max_samples > maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum > 0` and the `release == FALSE`.

- *RETCODE_NO_DATA* - no samples that meet the constraints are available.

3.5.2.73 **take_next_instance_w_condition**

Scope

SPACE::FooDataReader

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
        take_next_instance_w_condition
            (FooSeq& data_values,
             SampleInfoSeq& info_seq,
             Long max_samples,
             InstanceHandle_t a_handle,
             ReadCondition_ptr a_condition);
```

Description

This operation reads a sequence of *Foo* samples of the next single instance from the *FooDataReader*, filtered by a *ReadCondition* or *QueryCondition* and by doing so, removes the data from the *FooDataReader*.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned *SampleInfo* structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in InstanceHandle_t a_handle - the current single instance, the returned samples belong to the next single instance.

in ReadCondition_ptr a_condition - a pointer to a *ReadCondition* or *QueryCondition* which filters the data before it is returned by the read operation.

Return Value

ReturnCode_t - Possible return codes of the operation are: *RETCODE_OK*, *RETCODE_ERROR*, *RETCODE_BAD_PARAMETER*, *RETCODE_ALREADY_DELETED*, *RETCODE_OUT_OF_RESOURCES*, *RETCODE_NOT_ENABLED*, *RETCODE_PRECONDITION_NOT_MET* or *RETCODE_NO_DATA*.

Detailed Description

This operation reads a sequence of `Foo` samples of a single instance from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_next_instance_w_condition` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available and removed from the `FooDataReader`
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_condition` is not a valid `ReadCondition_ptr` or `a_handle` is not a valid handle
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `ReadCondition` or `QueryCondition` is not attached to this `FooDataReader`.
 - the `max_samples>maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum>0` and the `release==FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.74 take_next_sample

Scope

`SPACE::FooDataReader`

Synopsis

```
#include <ccpp_Space.h>
ReturnCode_t
take_next_sample
(Foo& data_value,
 SampleInfo sample_info);
```


Note: This operation is not yet implemented. It is scheduled for a future release.

3.5.2.75 **take_w_condition**

Scope

SPACE::FooDataReader

Synopsis

```
#include <ccpp_Space.h>
    ReturnCode_t
        take_w_condition
            (FooSeq& data_values,
             SampleInfoSeq& info_seq,
             Long max_samples,
             ReadCondition_ptr a_condition);
```

Description

This operation reads a sequence of Foo samples from the FooDataReader, filtered by a ReadCondition or QueryCondition and by doing so, removes the data from the FooDataReader.

Parameters

inout FooSeq& data_values - the returned sample data sequence. *data_values* is also used as an input to control the behaviour of this operation.

inout SampleInfoSeq& info_seq - the returned SampleInfo structure sequence. *info_seq* is also used as an input to control the behaviour of this operation.

in long max_samples - the maximum number of samples that is returned.

in ReadCondition_ptr a_condition - a pointer to a ReadCondition or QueryCondition which filters the data before it is returned by the read operation.

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED, RETCODE_OUT_OF_RESOURCES, RETCODE_NOT_ENABLED, RETCODE_PRECONDITION_NOT_MET or RETCODE_NO_DATA.

Detailed Description

This operation reads a sequence of `Foo` samples from the `FooDataReader`, filtered by a `ReadCondition` or `QueryCondition` and by doing so, removes the data from the `FooDataReader`, so it can not be read or taken again. The behaviour is identical to `read_w_condition` except for that the samples are removed from the `FooDataReader`.

Return Code

When the operation returns:

- `RETCODE_OK` - a sequence of data values is available and removed from the `FooDataReader`
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_BAD_PARAMETER` - the parameter `a_condition` is not a valid `ReadCondition_ptr`
- `RETCODE_ALREADY_DELETED` - the `FooDataReader` has already been deleted
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_NOT_ENABLED` - the `FooDataReader` is not enabled.
- `RETCODE_PRECONDITION_NOT_MET` - one of the following is true:
 - the `ReadCondition` or `QueryCondition` is not attached to this `FooDataReader`
 - the `max_samples>maximum` and `max_samples` is not `LENGTH_UNLIMITED`
 - one or more values of `length`, `maximum`, and `release` for the two sequences are not identical
 - the `maximum>0` and the `release==FALSE`.
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

3.5.2.76 `wait_for_historical_data` (inherited)

This operation is inherited and therefore not described here. See the class `DataReader` for further explanation.

Synopsis

```
#include <ccpp_Space.h>
      ReturnCode_t
      wait_for_historical_data
      (const Duration_t& max_wait);
```

3.5.3 Class DataSample

A `DataSample` represents an atom of data information (i.e. one value for an instance) as returned by the `DataReader`'s `read/take` operations. It consists of two parts: A `SampleInfo` and the Data itself. The Data part is the data as produced by a `Publisher`. The `SampleInfo` part contains additional information related to the data provided by the Data Distribution Service.

3.5.4 Struct SampleInfo

The struct `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

The interface description of this struct is as follows:

```
struct SampleInfo
{
    SampleStateKind sample_state;
    ViewStateKind view_state;
    InstanceStateKind instance_state;
    Time_t source_timestamp;
    InstanceHandle_t instance_handle;
    BuiltinTopicKey_t publication_handle;
    Long disposed_generation_count;
    Long no_writers_generation_count;
    Long sample_rank;
    Long generation_rank;
    Long absolute_generation_rank;
    Boolean valid_data;
};
```

The next paragraph describes the usage of the `SampleInfo` struct.

3.5.4.1 SampleInfo

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct SampleInfo
{
    SampleStateKind sample_state;
    ViewStateKind view_state;
    InstanceStateKind instance_state;
    Time_t source_timestamp;
    InstanceHandle_t instance_handle;
    BuiltinTopicKey_t publication_handle;
    Long disposed_generation_count;
    Long no_writers_generation_count;
```

```

    Long sample_rank;
    Long generation_rank;
    Long absolute_generation_rank;
    Boolean valid_data;
};

```

Description

The struct `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

Attributes

SampleStateKind sample_state - whether or not the corresponding data sample has already been read.

ViewStateKind view_state - whether the `DataReader` has already seen samples of the most-current generation of the related instance.

InstanceStateKind instance_state - whether the instance is alive, has no writers or is disposed of.

Time_t source_timestamp - the time provided by the `DataWriter` when the sample was written.

InstanceHandle_t instance_handle - the handle that identifies locally the corresponding instance.

InstanceHandle_t publication_handle - the handle that identifies locally the `DataWriter` that modified the instance. In fact it is the `instance_handle` of the builtin `DCPSPublication` sample that describes this `DataWriter`. It can be used as a parameter to the `DataReader` operation `get_matched_publication_data` to obtain this builtin `DCPSPublication` sample.

Long disposed_generation_count - the number of times the instance has become alive after it was disposed of explicitly by a `DataWriter`.

Long no_writers_generation_count - the number of times the instance has become alive after it was disposed of because there were no `DataWriter` objects.

Long sample_rank - the number of samples related to the same instance that are found in the collection returned by a `read` or `take` operation.

Long generation_rank - the generation difference between the time the sample was received and the time the most recent sample in the collection was received.

Long absolute_generation_rank - the generation difference between the time the sample was received and the time the most recent sample was received.

Boolean valid_data - whether the `DataSample` contains any meaningful data. If not, the sample is only used to communicate a change in the `instance_state` of the instance.

Detailed Description

The struct `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

Generations

A generation is defined as: ‘the number of times an instance has become alive (with `instance_state==ALIVE_INSTANCE_STATE`) at the time the sample was received’. Note that the generation counters are initialized to zero when a `DataReader` first detects a never-seen-before instance.

Two types of generations are distinguished: `disposed_generation_count` and `no_writers_generation_count`.

After a `DataWriter` disposes an instance, the `disposed_generation_count` for all `DataReaders` that already knew that instance will be incremented the next time the instance is written again.

If the `DataReader` detects that there are no live `DataWriter` entities, the `instance_state` of the `sample_info` will change from `ALIVE_INSTANCE_STATE` to `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`. The next time the instance is written, `no_writers_generation_count` will be incremented.

Sample Information

`SampleInfo` is the additional information that accompanies the data in each sample that is read or taken. It contains the following information:

- `sample_state` (`READ_SAMPLE_STATE` or `NOT_READ_SAMPLE_STATE`) indicates whether or not the corresponding data sample has already been read
- `view_state` (`NEW_VIEW_STATE`, or `NOT_NEW_VIEW_STATE`) indicates whether the `DataReader` has already seen samples of the most-current generation of the related instance
- `instance_state` (`ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`) indicates whether the instance is alive, has no writers or if it has been disposed of:
 - `ALIVE_INSTANCE_STATE` if this instance is currently in existence
 - `NOT_ALIVE_DISPOSED_INSTANCE_STATE` if this instance was disposed of by a `DataWriter`

- `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` none of the `DataWriter` objects currently “alive” (according to the `LivelinessQosPolicy`) are writing the instance.

- `source_timestamp` indicates the time provided by the `DataWriter` when the sample was written
- `instance_handle` indicates locally the corresponding instance
- `publication_handle` indicates system wide the publisher of the sample, the local publication handle can be found with the `DataReader` operation `lookup_instance` for the `publication_handle`
- `disposed_generation_count` indicates the number of times the instance has become alive after it was disposed of explicitly by a `DataWriter`, at the time the sample was received
- `no_writers_generation_count` indicates the number of times the instance has become alive after its `instance_state` has been `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`, at the time the sample was received
- `sample_rank` indicates the number of samples related to the same instance that follow in the collection returned by a `read` or `take` operation
- `generation_rank` indicates the generation difference (number of times the instance was disposed of and become alive again) between the time the sample was received and the time the most recent sample in the collection (related to the same instance) was received
- `absolute_generation_rank` indicates the generation difference (number of times the instance was disposed of and become alive again) between the time the sample was received and the time the most recent sample (which may not be in the returned collection), related to the same instance, was received.
- `valid_data` indicates whether the corresponding data value contains any meaningful data. If not, the data value is just a ‘dummy’ sample for which only the keyfields have been assigned. It is used to accompany the `SampleInfo` that communicates a change in the `instance_state` of an instance for which there is no ‘real’ sample available.

3.5.5 SubscriberListener Interface

Since a `Subscriber` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `SubscriberListener`. This interface must be implemented by the application. A user defined class must be provided by the application which must extend from the

SubscriberListener class. All SubscriberListener operations must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.



All operations for this interface must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

The SubscriberListener provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a QosPolicy setting, etc. The SubscriberListener is related to changes in communication status.

The interface description of this class is as follows:

```
class SubscriberListener
{
//
// inherited from class DataReaderListener
//
// void
//   on_requested_deadline_missed
//     (DataReader_ptr reader,
//      const RequestedDeadlineMissedStatus& status) = 0;

// void
//   on_requested_incompatible_qos
//     (DataReader_ptr reader,
//      const RequestedIncompatibleQosStatus& status) = 0;

// void
//   on_sample_rejected
//     (DataReader_ptr reader,
//      const SampleRejectedStatus& status) = 0;

// void
//   on_liveliness_changed
//     (DataReader_ptr reader,
//      const LivelinessChangedStatus& status) = 0;

// void
//   on_data_available
//     (DataReader_ptr reader) = 0;

// void
//   on_subscription_matched
//     (DataReader_ptr reader,
//      const SubscriptionMatchedStatus& status) = 0;
```

```

// void
//     on_sample_lost
//         (DataReader_ptr reader,
//          const SampleLostStatus& status) = 0;
//
// abstract external operations
//
//     void
//         on_data_on_readers
//             (Subscriber_ptr subs) = 0;
//
// implemented API operations
//     <no operations>
//
};

```

The next paragraphs list all `SubscriberListener` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited. The abstract operation is fully described since it must be implemented by the application.

3.5.5.1 `on_data_available` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_data_available
        (DataReader_ptr reader) = 0;

```

3.5.5.2 `on_data_on_readers` (abstract)

Scope

```
DDS::SubscriberListener
```

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_data_on_readers
        (Subscriber_ptr subs) = 0;

```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when new data is available.

Parameters

in Subscriber_ptr subs - contain a pointer to the Subscriber for which data is available (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when new data is available for this Subscriber. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `SubscriberListener` is installed and enabled for the `DATA_ON_READERS_STATUS`.

The Data Distribution Service will provide a reference to the Subscriber in the parameter `subs` for use by the application.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` will occur together. In case these status changes occur, the Data Distribution Service will look for an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look for an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

Note that if `on_data_on_readers` is called, then the Data Distribution Service will not try to call `on_data_available`, however, the application can force a call to the callback function `on_data_available` of `DataReaderListener` objects that have data by means of the `notify_datareaders` operation.

3.5.5.3 `on_liveliness_changed` (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_liveliness_changed
        (DataReader_ptr reader,
         const LivelinessChangedStatus& status) = 0;
```

3.5.5.4 on_requested_deadline_missed (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_requested_deadline_missed
        (DataReader_ptr reader,
         const RequestedDeadlineMissedStatus& status) = 0;
```

3.5.5.5 on_requested_incompatible_qos (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_requested_incompatible_qos
        (DataReader_ptr reader,
         const RequestedIncompatibleQosStatus& status) = 0;
```

3.5.5.6 on_sample_lost (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_sample_lost
        (DataReader_ptr reader,
         const SampleLostStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.5.5.7 on_sample_rejected (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_sample_rejected
        (DataReader_ptr reader,
         const SampleRejectedStatus& status) = 0;
```

3.5.5.8 on_subscription_matched (inherited, abstract)

This operation is inherited and therefore not described here. See the class `DataReaderListener` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_subscription_matched
        (DataReader_ptr reader,
         const SubscriptionMatchedStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.5.6 DataReaderListener Interface

Since a `DataReader` is an `Entity`, it has the ability to have a `Listener` associated with it. In this case, the associated `Listener` should be of type `DataReaderListener`. This interface must be implemented by the application. A user defined class must be provided by the application which must extend from the `DataReaderListener` class. **All** `DataReaderListener` operations **must** be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.



All operations for this interface must be implemented in the user defined class, it is up to the application whether an operation is empty or contains some functionality.

The `DataReaderListener` provides a generic mechanism (actually a callback function) for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a `QosPolicy` setting, etc. The `DataReaderListener` is related to changes in communication status.

The interface description of this class is as follows:

```
class DataReaderListener
{
//
// abstract external operations
//
void
    on_requested_deadline_missed
        (DataReader_ptr reader,
         const RequestedDeadlineMissedStatus& status) = 0;
void
    on_requested_incompatible_qos
        (DataReader_ptr reader,
         const RequestedIncompatibleQosStatus& status) = 0;
```

```

void
    on_sample_rejected
        (DataReader_ptr reader,
         const SampleRejectedStatus& status) = 0;

void
    on_liveliness_changed
        (DataReader_ptr reader,
         const LivelinessChangedStatus& status) = 0;

void
    on_data_available
        (DataReader_ptr reader) = 0;

void
    on_subscription_matched
        (DataReader_ptr reader,
         const SubscriptionMatchedStatus& status) = 0;

void
    on_sample_lost
        (DataReader_ptr reader,
         const SampleLostStatus& status) = 0;
//
//  implemented API operations
//      <no operations>
//
};

```

The next paragraphs describe the usage of all `DataReaderListener` operations. These abstract operations are fully described because they must be implemented by the application.

3.5.6.1 `on_data_available` (abstract)

Scope

`DDS::DataReaderListener`

Synopsis

```

#include <ccpp_dds_dcps.h>
void
    on_data_available
        (DataReader_ptr reader) = 0;

```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when new data is available.

Parameters

in DataReader_ptr reader - contain a pointer to the `DataReader` for which data is available (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when new data is available for this `DataReader`. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `DATA_AVAILABLE_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` for use by the application.

The statuses `DATA_ON_READERS_STATUS` and `DATA_AVAILABLE_STATUS` will occur together. In case these status changes occur, the Data Distribution Service will look for an attached and activated `SubscriberListener` or `DomainParticipantListener` (in that order) for the `DATA_ON_READERS_STATUS`. In case the `DATA_ON_READERS_STATUS` can not be handled, the Data Distribution Service will look for an attached and activated `DataReaderListener`, `SubscriberListener` or `DomainParticipantListener` for the `DATA_AVAILABLE_STATUS` (in that order).

Note that if `on_data_on_readers` is called, then the Data Distribution Service will not try to call `on_data_available`, however, the application can force a call to the `DataReader` objects that have data by means of the `notify_datareaders` operation.

3.5.6.2 on_liveliness_changed (abstract)

Scope

`DDS::DataReaderListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_liveliness_changed
        (DataReader_ptr reader,
         const LivelinessChangedStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the liveliness of one or more `DataWriter` objects that were writing instances read through this `DataReader` has changed.

Parameters

in `DataReader_ptr reader` - contain a pointer to the `DataReader` for which the liveliness of one or more `DataWriter` objects has changed (this is an input to the application provided by the Data Distribution Service).

in `const LivelinessChangedStatus& status` - contain the `LivelinessChangedStatus` struct (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the liveliness of one or more `DataWriter` objects that were writing instances read through this `DataReader` has changed. In other words, some `DataWriter` have become “alive” or “not alive”. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `LIVELINESS_CHANGED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `LivelinessChangedStatus` struct for use by the application.

3.5.6.3 on_requested_deadline_missed (abstract)

Scope

`DDS::DataReaderListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_requested_deadline_missed
        (DataReader_ptr reader,
         const RequestedDeadlineMissedStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected.

Parameters

in `DataReader_ptr reader` - contain a pointer to the `DataReader` for which the deadline was missed (this is an input to the application provided by the Data Distribution Service).

in `const RequestedDeadlineMissedStatus& status` - contain the `RequestedDeadlineMissedStatus` struct (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the deadline that the `DataReader` was expecting through its `DeadlineQosPolicy` was not respected for a specific instance. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `REQUESTED_DEADLINE_MISSED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `RequestedDeadlineMissedStatus` struct in the parameter `status` for use by the application.

3.5.6.4 on_requested_incompatible_qos (abstract)

Scope

`DDS::DataReaderListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_requested_incompatible_qos
        (DataReader_ptr reader,
         const RequestedIncompatibleQosStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `REQUESTED_INCOMPATIBLE_QOS_STATUS` changes.

Parameters

in `DataReader_ptr reader` - a pointer to the `DataReader` provided by the Data Distribution Service.

in `const RequestedIncompatibleQosStatus& status` - the `RequestedIncompatibleQosStatus` struct provided by the Data Distribution Service.

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when the `REQUESTED_INCOMPATIBLE_QOS_STATUS` changes. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `DataReaderListener` is installed and enabled for the `REQUESTED_INCOMPATIBLE_QOS_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `RequestedIncompatibleQosStatus` struct in the parameter `status`, for use by the application.

The application can use this operation as a callback function implementing a proper response to the status change. This operation is enabled by setting the `REQUESTED_INCOMPATIBLE_QOS_STATUS` in the mask in the call to `DataReader::set_listener`. When the `DataReaderListener` on the `DataReader` is not enabled for the `REQUESTED_INCOMPATIBLE_QOS_STATUS`, the status change will propagate to the `SubscriberListener` of the `Subscriber` (if enabled) or to the `DomainParticipantListener` of the `DomainParticipant` (if enabled).

3.5.6.5 on_sample_lost (abstract)

Scope

`DDS::DataReaderListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_sample_lost
        (DataReader_ptr reader,
         const SampleLostStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.5.6.6 on_sample_rejected (abstract)**Scope**

DDS::DataReaderListener

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_sample_rejected
        (DataReader_ptr reader,
         const SampleRejectedStatus& status) = 0;
```

Description

This operation must be implemented by the application and is called by the Data Distribution Service when a sample has been rejected.

Parameters

in DataReader_ptr reader - contain a pointer to the DataReader for which a sample has been rejected (this is an input to the application provided by the Data Distribution Service).

in const SampleRejectedStatus& status - contain the SampleRejectedStatus struct (this is an input to the application provided by the Data Distribution Service).

Return Value

<none>

Detailed Description

This operation is the external operation (interface, which must be implemented by the application) that is called by the Data Distribution Service when a (received) sample has been rejected. The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant DataReaderListener is installed and enabled for the SAMPLE_REJECTED_STATUS.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `SampleRejectedStatus` struct in the parameter `status` for use by the application.

3.5.6.7 on_subscription_matched (abstract)

Scope

`DDS::DataReaderListener`

Synopsis

```
#include <ccpp_dds_dcps.h>
void
    on_subscription_matched
        (DataReader_ptr reader,
         const SubscriptionMatchedStatus& status) = 0;
```

Note: This operation is not yet supported. It is scheduled for a future release.

3.5.7 Class ReadCondition

The `DataReader` objects can create a set of `ReadCondition` (and `StatusCondition`) objects which provide support (in conjunction with `WaitSet` objects) for an alternative communication style between the Data Distribution Service and the application (i.e., state-based rather than event-based).

`ReadCondition` objects allow an `DataReader` to specify the data samples it is interested in (by specifying the desired sample-states, view-states, and instance-states); see the parameter definitions for `DataReader`'s `create_readcondition` operation. This allows the Data Distribution Service to trigger the condition only when suitable information is available. `ReadCondition` objects are to be used in conjunction with a `WaitSet`. More than one `ReadCondition` may be attached to the same `DataReader`.

The interface description of this class is as follows:

```
class ReadCondition
{
//
// inherited from Condition
//
// Boolean
//     get_trigger_value
//     (void);
//
// implemented API operations
//
    SampleStateMask
        get_sample_state_mask
            (void);
```

```

ViewStateMask
    get_view_state_mask
        (void);

InstanceStateMask
    get_instance_state_mask
        (void);

DataReader_ptr
    get_datareader
        (void);
};

```

The next paragraphs describe the usage of all `ReadCondition` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.7.1 `get_datareader`

Scope

`DDS::ReadCondition`

Synopsis

```

#include <ccpp_dds_dcps.h>
DataReader_ptr
    get_datareader
        (void);

```

Description

This operation returns the `DataReader` associated with the `ReadCondition`.

Parameters

<none>

Return Value

DataReader_ptr - Result value is a pointer to the `DataReader`.

Detailed Description

This operation returns the `DataReader` associated with the `ReadCondition`. Note that there is exactly one `DataReader` associated with each `ReadCondition` (i.e. the `DataReader` that created the `ReadCondition` object).

3.5.7.2 `get_instance_state_mask`

Scope

DDS::ReadCondition

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceStateMask
    get_instance_state_mask
        (void);
```

Description

This operation returns the set of `instance_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

Parameters

<none>

Return Value

InstanceStateMask - Result value are the `instance_states` specified when the `ReadCondition` was created.

Detailed Description

This operation returns the set of `instance_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

The `instance_states` returned are the `instance_states` specified when the `ReadCondition` was created. `instance_states` can be `ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE`, `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or a combination of these.

3.5.7.3 `get_sample_state_mask`

Scope

DDS::ReadCondition

Synopsis

```
#include <ccpp_dds_dcps.h>
SampleStateMask
    get_sample_state_mask
        (void);
```

Description

This operation returns the set of `sample_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

Parameters

<none>

Return Value

SampleStateMask - Result value are the `sample_states` specified when the `ReadCondition` was created.

Detailed Description

This operation returns the set of `sample_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

The `sample_states` returned are the `sample_states` specified when the `ReadCondition` was created. `sample_states` can be `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE` or both.

3.5.7.4 `get_trigger_value` (inherited)

This operation is inherited and therefore not described here. See the class `Condition` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
Boolean
    get_trigger_value
        (void);
```

3.5.7.5 `get_view_state_mask`**Scope**

`DDS::ReadCondition`

Synopsis

```
#include <ccpp_dds_dcps.h>
ViewStateMask
    get_view_state_mask
        (void);
```

Description

This operation returns the set of `view_states` that are taken into account to determine the `trigger_value` of the `ReadCondition`.

Parameters

<none>

Return Value

ViewStateMask - Result value are the *view_states* specified when the *ReadCondition* was created.

Detailed Description

This operation returns the set of *view_states* that are taken into account to determine the *trigger_value* of the *ReadCondition*.

The *view_states* returned are the *view_states* specified when the *ReadCondition* was created. *view_states* can be *NEW_VIEW_STATE*, *NOT_NEW_VIEW_STATE* or both.

3.5.8 Class QueryCondition

QueryCondition objects are specialized *ReadCondition* objects that allow the application to specify a filter on the locally available data. The *DataReader* objects accept a set of *QueryCondition* objects for the *DataReader* and provide support (in conjunction with *WaitSet* objects) for an alternative communication style between the Data Distribution Service and the application (i.e., state-based rather than event-based).

Query Function

QueryCondition objects allow an application to specify the data samples it is interested in (by specifying the desired sample-states, view-states, instance-states and query expression); see the parameter definitions for *DataReader*'s *read/take* operations. This allows the Data Distribution Service to trigger the condition only when suitable information is available. *QueryCondition* objects are to be used in conjunction with a *WaitSet*. More than one *QueryCondition* may be attached to the same *DataReader*.

The query (*query_expression*) is similar to an SQL WHERE clause and can be parameterized by arguments that are dynamically changeable with the *set_query_parameters* operation.

The interface description of this class is as follows:

```

class QueryCondition
{
//
// inherited from ReadCondition
//
// SampleStateMask
//     get_sample_state_mask

```

```

//      (void);

// ViewStateMask
//      get_view_state_mask
//      (void);

// InstanceStateMask
//      get_instance_state_mask
//      (void);

// DataReader_ptr
//      get_datareader
//      (void);
// Boolean
//      get_trigger_value
//      (void);
//
// implemented API operations
//
char*
    get_query_expression
        (void);

ReturnCode_t
    get_query_parameters
        (StringSeq& query_parameters);

ReturnCode_t
    set_query_parameters
        (const StringSeq& query_parameters);
};

```

The next paragraphs describe the usage of all `QueryCondition` operations. The inherited operations are listed but not fully described because they are not implemented in this class. The full description of these operations is given in the classes from which they are inherited.

3.5.8.1 `get_datareader` (inherited)

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```

#include <ccpp_dds_dcps.h>
DataReader_ptr
    get_datareader
        (void);

```

3.5.8.2 `get_instance_state_mask` (inherited)

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
InstanceStateMask
    get_instance_state_mask
        (void);
```

3.5.8.3 `get_query_parameters`

Scope

`DDS::QueryCondition`

Synopsis

```
#include <ccpp_dds_dcps.h>
ReturnCode_t
    get_query_parameters
        (StringSeq& query_parameters);
```

Description

This operation obtains the `query_parameters` associated with the `QueryCondition`.

Parameters

inout StringSeq& query_parameters - a reference to a sequence of strings that will be used to store the parameters used in the SQL expression.

Return Value

ReturnCode_t - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED` or `RETCODE_OUT_OF_RESOURCES`.

Detailed Description

This operation obtains the `query_parameters` associated with the `QueryCondition`. That is, the parameters specified on the last successful call to `set_query_arguments` or, if `set_query_arguments` was never called, the arguments specified when the `QueryCondition` were created.

The resulting handle contains a sequence of strings with the parameters used in the SQL expression (i.e., the %n tokens in the expression). The number of parameters in the result sequence will exactly match the number of %n tokens in the query expression associated with the `QueryCondition`.

Return Code

When the operation returns:

- `RETCODE_OK` - the existing set of query parameters applied to this `QueryCondition` has successfully been copied into the specified `query_parameters` parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `QueryCondition` has already been deleted.
- `RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.

3.5.8.4 `get_query_expression`

Scope

`DDS::QueryCondition`

Synopsis

```
#include <ccpp_dds_dcps.h>
char*
    get_query_expression
        (void);
```

Description

This operation returns the query expression associated with the `QueryCondition`.

Parameters

<none>

Return Value

`char*` - Result value is a reference to the query expression associated with the `QueryCondition`.

Detailed Description

This operation returns the query expression associated with the `QueryCondition`. That is, the expression specified when the `QueryCondition` was created. The operation will return `NULL` when there was an internal error or when the `QueryCondition` was already deleted. If there were no parameters, an empty sequence is returned.

3.5.8.5 `get_sample_state_mask` (inherited)

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
SampleStateMask
    get_sample_state_mask
        (void);
```

3.5.8.6 `get_trigger_value` (inherited)

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
Boolean
    get_trigger_value
        (void);
```

3.5.8.7 `get_view_state_mask` (inherited)

This operation is inherited and therefore not described here. See the class `ReadCondition` for further explanation.

Synopsis

```
#include <ccpp_dds_dcps.h>
ViewStateMask
    get_view_state_mask
        (void);
```

3.5.8.8 `set_query_parameters`

Scope

`DDS::QueryCondition`

Synopsis

```
#include <ccpp_dds_dcps.h>
```

```

ReturnCode_t
    set_query_arguments
        (const StringSeq& parameters);

```

Description

This operation changes the query parameters associated with the QueryCondition.

Parameters

in const StringSeq& query_parameters - a sequence of strings which are the parameters used in the SQL query string (i.e., the “%n” tokens in the expression).

Return Value

ReturnCode_t - Possible return codes of the operation are: RETCODE_OK, RETCODE_ERROR, RETCODE_BAD_PARAMETER, RETCODE_ALREADY_DELETED or RETCODE_OUT_OF_RESOURCES.

Detailed Description

This operation changes the query parameters associated with the QueryCondition. The parameter *query_parameters* is a sequence of strings which are the parameter values used in the SQL query string (i.e., the “%n” tokens in the expression). The number of values in *query_parameters* must be equal or greater than the highest referenced %n token in the *query_expression* (e.g. if %1 and %8 are used as parameter in the *query_expression*, the *query_parameters* should at least contain $n+1 = 9$ values).

Return Code

When the operation returns:

- *RETCODE_OK* - the query parameters associated with the QueryCondition are changed.
- *RETCODE_ERROR* - an internal error has occurred.
- *RETCODE_BAD_PARAMETER* - the number of parameters in *query_parameters* does not match the number of “%n” tokens in the expression for this QueryCondition or one of the parameters is an illegal parameter.
- *RETCODE_ALREADY_DELETED* - the QueryCondition has already been deleted.
- *RETCODE_OUT_OF_RESOURCES* - the Data Distribution Service ran out of resources to complete this operation.

A close-up, low-angle photograph of a computer keyboard, focusing on the central and lower-right keys. The keys are white with dark lettering. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

APPENDICES



Quality Of Service

Each Entity is accompanied by an `<Entity>Qos` structure that implements the basic mechanism for an application to specify Quality of Service attributes. This structure consists of Entity specific `QosPolicy` attributes. `QosPolicy` attributes are structured types where each type specifies the information that controls an Entity related (configurable) attribute of the Data Distribution Service. A `QosPolicy` attribute struct is identified as `<name>QosPolicy`.

Affected Entities

Each Entity can be configured with a set of `QosPolicy` settings. However, any Entity cannot support any `QosPolicy`. For instance, a `DomainParticipant` supports different `QosPolicy` settings than a `Topic` or a `Publisher`. The set of `QosPolicy` settings is implemented as a struct of `QosPolicy` structs, identified as `<Entity>Qos`. Each `<Entity>Qos` struct only contains those `QosPolicy` structs relevant to the specific Entity. The `<Entity>Qos` struct serves as the parameter to operations which require a `Qos`. `<Entity>Qos` struct is the API implementation of the QoS. Depending on the specific `<Entity>Qos`, it controls the behaviour of a `Topic`, `DataWriter`, `DataReader`, `Publisher`, `Subscriber`, `DomainParticipant` or `DomainParticipantFactory`¹.

Basic Usage

The basic way to modify or set the `<Entity>Qos` is by using an `get_qos` operation to get all `QosPolicy` settings from this Entity (that is the `<Entity>Qos`), modify several specific `QosPolicy` settings and put them back using an `set_qos` operation to set all `QosPolicy` settings on this Entity (that is the `<Entity>Qos`). An example of these operations for the `DataWriterQos` are `get_default_datawriter_qos` and `set_default_datawriter_qos`, which take the `DataWriterQos` as a parameter.

The interface description of this struct is as follows:

```
// struct <name>QosPolicy
//     see appendix
//
```

-
1. Note that the `DomainParticipantFactory` is a special kind of entity: it does not inherit from `Entity`, nor does it have a `Listener` or `StatusCondition`, but its behaviour can be controlled by its own set of `QosPolicies`.

```
//
// struct <Entity>Qos
//
    struct DomainParticipantFactoryQos
    { EntityFactoryQosPolicy          entity_factory; };
    struct DomainParticipantQos
    { UserDataQosPolicy              user_data;
      EntityFactoryQosPolicy          entity_factory;
      SchedulingQosPolicy             watchdog_scheduling;
      SchedulingQosPolicy             listener_scheduling; };
    struct TopicQos
    { TopicDataQosPolicy              topic_data;
      DurabilityQosPolicy             durability;
      DurabilityServiceQosPolicy       durability_service;
      DeadlineQosPolicy               deadline;
      LatencyBudgetQosPolicy           latency_budget;
      LivelinessQosPolicy             liveliness;
      ReliabilityQosPolicy             reliability;
      DestinationOrderQosPolicy        destination_order;
      HistoryQosPolicy                history;
      ResourceLimitsQosPolicy          resource_limits;
      TransportPriorityQosPolicy        transport_priority;
      LifespanQosPolicy               lifespan;
      OwnershipQosPolicy              ownership; };
    struct DataWriterQos
    { DurabilityQosPolicy             durability;
      DeadlineQosPolicy              deadline;
      LatencyBudgetQosPolicy          latency_budget;
      LivelinessQosPolicy             liveliness;
      ReliabilityQosPolicy            reliability;
      DestinationOrderQosPolicy        destination_order;
      HistoryQosPolicy               history;
      ResourceLimitsQosPolicy          resource_limits;
      TransportPriorityQosPolicy        transport_priority;
      LifespanQosPolicy              lifespan;
      UserDataQosPolicy              user_data;
      OwnershipQosPolicy             ownership;
      OwnershipStrengthQosPolicy       ownership_strength;
      WriterDataLifecycleQosPolicy     writer_data_lifecycle; };
    struct PublisherQos
    { PresentationQosPolicy           presentation;
      PartitionQosPolicy              partition;
      GroupDataQosPolicy              group_data;
      EntityFactoryQosPolicy          entity_factory; };
    struct DataReaderQos
    { DurabilityQosPolicy             durability;
      DeadlineQosPolicy              deadline;
      LatencyBudgetQosPolicy          latency_budget;
      LivelinessQosPolicy            liveliness;
      ReliabilityQosPolicy            reliability;
```



```

        DestinationOrderQosPolicy    destination_order;
        HistoryQosPolicy              history;
        ResourceLimitsQosPolicy       resource_limits;
        UserDataQosPolicy             user_data;
        OwnershipQosPolicy            ownership;
        TimeBasedFilterQosPolicy       time_based_filter;
        ReaderDataLifecycleQosPolicy   reader_data_lifecycle;};

struct SubscriberQos
{
    PresentationQosPolicy    presentation;
    PartitionQosPolicy       partition;
    GroupDataQosPolicy       group_data;
    EntityFactoryQosPolicy   entity_factory; };

//
// define <Entity>_QOS_DEFAULT
//
#define PARTICIPANT_QOS_DEFAULT
#define TOPIC_QOS_DEFAULT
#define DATAWRITER_QOS_DEFAULT
#define PUBLISHER_QOS_DEFAULT
#define DATAREADER_QOS_DEFAULT
#define SUBSCRIBER_QOS_DEFAULT
#define DATAWRITER_QOS_USE_TOPIC_QOS
#define DATAREADER_QOS_USE_TOPIC_QOS
//
// implemented API operations
//   <no operations>
//

```

The next paragraphs describe the usage of each <Entity>Qos struct.

DataReaderQos

Scope

DDS

Synopsis

```

#include <ccpp_dds_dcps.h>
struct DataReaderQos
struct DataReaderQos
{
    DurabilityQosPolicy    durability;
    DeadlineQosPolicy      deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy    liveliness;
    ReliabilityQosPolicy    reliability;
    DestinationOrderQosPolicy destination_order;
    HistoryQosPolicy        history;
    ResourceLimitsQosPolicy resource_limits;
    UserDataQosPolicy       user_data;
}

```

```
OwnershipQosPolicy      ownership;
TimeBasedFilterQosPolicy time_based_filter;
ReaderDataLifecycleQosPolicy reader_data_lifecycle;};
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a `DataReader`.

Attributes

DurabilityQosPolicy durability - whether the data should be stored for late joining- *deadline* - the period within which a new sample is expected.

LatencyBudgetQosPolicy latency_budget - used by the Data Distribution Service for optimization.

LivelinessQosPolicy liveliness - the way the liveliness of the `DataReader` is asserted to the Data Distribution Service.

ReliabilityQosPolicy reliability - the reliability of the data distribution.

DestinationOrderQosPolicy destination_order - the order in which the `DataReader` timely orders the data.

HistoryQosPolicy history - how samples should be stored.

ResourceLimitsQosPolicy resource_limits - the maximum amount of resources to be used.

UserDataQosPolicy user_data - used to attach additional information to the `DataReader`.

OwnershipQosPolicy ownership - whether a `DataWriter` exclusively owns an instance.

TimeBasedFilterQosPolicy time_based_filter - the maximum data rate at which the `DataReader` will receive changes.

ReaderDataLifecycleQosPolicy reader_data_lifecycle - the minimum time after which a data instance is disposed of when the `instance_state` has become either `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` or `NOT_ALIVE_DISPOSED_INSTANCE_STATE`.

Detailed Description

A `QosPolicy` can be set when the `DataReader` is created with the `create_datareader` operation (or modified with the `set_qos` operation). Both operations take the `DataReaderQos` struct as a parameter. There may be cases

where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some `QosPolicy` have “immutable” semantics meaning that they can only be specified either at `DataReader` creation time or prior to calling the enable operation on the `DataReader`.

The initial value of the default `DataReaderQos` in the `Subscriber` are given in the following table:

Table 15 QosPolicy Values

QosPolicy	Field	Value
durability	kind	VOLATILE_DURABILITY_QOS
deadline	period	DURATION_INFINITE
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITE
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	FALSE
destination_order	kind	BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
user_data	value.length	0
ownership	kind	SHARED_OWNERSHIP_QOS
time_based_filter	minimum_separation	0
reader_data_lifecycle	autopurge_nowriter_samples_delay	DURATION_INFINITE
	autopurge_disposed_samples_delay	DURATION_INFINITE

DataWriterQos

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct DataWriterQos
{
    DurabilityQosPolicy      durability;
    DeadlineQosPolicy        deadline;
    LatencyBudgetQosPolicy   latency_budget;
    LivelinessQosPolicy      liveliness;
    ReliabilityQosPolicy     reliability;
    DestinationOrderQosPolicy destination_order;
    HistoryQosPolicy          history;
    ResourceLimitsQosPolicy   resource_limits;
    TransportPriorityQosPolicy transport_priority;
    LifespanQosPolicy          lifespan;
    UserDataQosPolicy         user_data;
    OwnershipQosPolicy        ownership;
    OwnershipStrengthQosPolicy ownership_strength;
    WriterDataLifecyleQosPolicy writer_data_lifecycle;};
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a DataWriter.

Attributes

DurabilityQosPolicy durability - whether the data should be stored for late joining readers.

DeadlineQosPolicydeadline - the period within which a new sample is written.

LatencyBudgetQosPolicy latency_budget - used by the Data Distribution Service for optimization.

LivelinessQosPolicy liveliness - the way the liveliness of the DataWriter is asserted to the Data Distribution Service.

ReliabilityQosPolicy reliability - the reliability of the data distribution.

DestinationOrderQosPolicydestination_order - the order in which the DataReader timely orders the data.

HistoryQosPolicy history - how samples should be stored.

ResourceLimitsQosPolicy resource_limits - the maximum amount of resources to be used.

TransportPriorityQosPolicy transport_priority - a priority hint for the underlying transport layer.

LifespanQosPolicylifespan - the maximum duration of validity of the data written by the DataWriter.

UserDataQosPolicy *user_data* - used to attach additional information to the *DataWriter*.

OwnershipQosPolicy *ownership* - whether a *DataWriter* exclusively owns an instance.

OwnershipStrengthQosPolicy *ownership_strength* - the strength to determine the ownership.

WriterDataLifecycleQosPolicy *writer_data_lifecycle* - whether unregistered instances are disposed of automatically or not.

Detailed Description

A *QosPolicy* can be set when the *DataWriter* is created with the *create_datawriter* operation (or modified with the *set_qos* operation). Both operations take the *DataWriterQos* struct as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the *set_qos* operation.

Some *QosPolicy* have “immutable” semantics meaning that they can only be specified either at *DataWriter* creation time or prior to calling the *enable* operation on the *DataWriter*.

The initial value of the default *DataWriterQos* in the *Publisher* are given in the following table:

Table 16 DATAWRITER_QOS_DEFAULT

QosPolicy	Field	Value
durability	kind	VOLATILE_DURABILITY_QOS
deadline	period	DURATION_INFINITE
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITE
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	FALSE
destination_order	kind	BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1

Table 16 DATAWRITER_QOS_DEFAULT (Continued)

QosPolicy	Field	Value
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
transport_priority	value	0
lifespan	duration	DURATION_INFINITE
user_data	value.length	0
ownership	kind	SHARED_OWNERSHIP_QOS
ownership_strength	value	0
writer_data_lifecycle	autodispose_ unregistered_instances	TRUE

DomainParticipantFactoryQos

Synopsis

```
#include <ccpp_dds_dcps.h>
struct DomainParticipantFactoryQos
    { EntityFactoryQosPolicy entity_factory; };
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a DomainParticipantFactory.

Attributes

EntityFactoryQosPolicy entity_factory - whether a just created DomainParticipant should be enabled.

Detailed Description

The QosPolicy cannot be set at creation time, since the DomainParticipantFactory is a pre-existing object that can only be obtained with the DomainParticipantFactory::get_instance operation or its alias TheParticipantFactory. Therefore its QosPolicy is initialized to a default value according to the following table:

Table 17 Default values for DomainParticipantFactoryQos

QosPolicy	Attribute	Value
entity_factory	autoenable_created_entities	TRUE

After creation the `QosPolicy` can be modified with the `set_qos` operation on the `DomainParticipantFactory`, which takes the `DomainParticipantFactoryQos` struct as a parameter.

DomainParticipantQos

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct DomainParticipantQos
{
    UserDataQosPolicy          user_data;
    EntityFactoryQosPolicy    entity_factory;
    SchedulingQosPolicy        watchdog_scheduling;
    SchedulingQosPolicy        listener_scheduling;
};
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a `DomainParticipant`.

Attributes

UserDataQosPolicy user_data - used to attach additional information to the `DomainParticipant`.

EntityFactoryQosPolicy entity_factory - whether a just created `Entity` should be enabled.

SchedulingQosPolicy watchdog_scheduling - the scheduling parameters used to create the watchdog thread.

SchedulingQosPolicy listener_scheduling - the scheduling parameters used to create the listener thread.

Detailed Description

A `DomainParticipant` will spawn different threads for different purposes:

- A listener thread is spawned to perform the callbacks to all `Listener` objects attached to the various `Entities` contained in the `DomainParticipant`. The scheduling parameters for this thread can be specified in the `listener_scheduling` field of the `DomainParticipantQos`.

- A watchdog thread is spawned to report the the Liveliness of all Entities contained in the DomainParticipant whose LivelinessQosPolicyKind in their LivelinessQosPolicy is set to AUTOMATIC_LIVELINESS_QOS. The scheduling parameters for this thread can be specified in the watchdog_scheduling field of the DomainParticipantQos.

A QosPolicy can be set when the DomainParticipant is created with the create_participant operation (or modified with the set_qos operation). Both operations take the DomainParticipantQos struct as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the set_qos operation.

Some QosPolicy have “immutable” semantics meaning that they can only be specified either at DomainParticipant creation time or prior to calling the enable operation on the DomainParticipant.

The initial value of the default DomainParticipantQos in the DomainParticipantFactory are given in the following table:

Table 18 PARTICIPANT_QOS_DEFAULT

QosPolicy	Field	Value
user_data	value.length	0
entity_factory	autoenable_created_entities	True
watchdog_scheduling	scheduling_class.kind	SCHEDULE_DEFAULT
	scheduling_priority_kind.kind	PRIORITY_RELATIVE
	scheduling_priority	0
listener_scheduling	scheduling_class.kind	SCHEDULE_DEFAULT
	scheduling_priority_kind.kind	PRIORITY_RELATIVE
	scheduling_priority	0

PublisherQos

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct PublisherQos
{
    PresentationQosPolicy    presentation;
    PartitionQosPolicy       partition;
    GroupDataQosPolicy       group_data;
```



```
EntityFactoryQosPolicy  entity_factory; };
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a Publisher.

Attributes

PresentationQosPolicy presentation - the dependency of changes to data-instances.

PartitionQosPolicy partition - the partitions in which the Publisher is active.

GroupDataQosPolicy group_data - used to attach additional information to the Publisher.

EntityFactoryQosPolicy entity_factory - whether a just created DataWriter should be enabled.

Detailed Description

A QosPolicy can be set when the Publisher is created with the `create_publisher` operation (or modified with the `set_qos` operation). Both operations take the `PublisherQos` struct as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some QosPolicy have “immutable” semantics meaning that they can only be specified either at Publisher creation time or prior to calling the `enable` operation on the Publisher.

The initial value of the default `PublisherQos` in the `DomainParticipant` are given in the following table:

Table 19 PUBLISHER_QOS_DEFAULT

QosPolicy	Field	Value
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	FALSE
	ordered_access	FALSE
partition	name.length	0
group_data	value.length	0
entity_factory	autoenable_created_entities	TRUE

SubscriberQos

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct SubscriberQos
{
    PresentationQosPolicy    presentation;
    PartitionQosPolicy       partition;
    GroupDataQosPolicy       group_data;
    EntityFactoryQosPolicy   entity_factory; };
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a Subscriber.

Attributes

PresentationQosPolicy presentation - the dependency of changes to data-instances.

PartitionQosPolicy partition - the partitions in which the Subscriber is active.

GroupDataQosPolicy group_data - used to attach additional information to the Subscriber.

EntityFactoryQosPolicy entity_factory - whether a just created DataReader should be enabled.

Detailed Description

A *QosPolicy* can be set when the Subscriber is created with the `create_subscriber` operation (or modified with the `set_qos` operation). Both operations take the *SubscriberQos* struct as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some *QosPolicy* have “immutable” semantics meaning that they can only be specified either at Subscriber creation time or prior to calling the `enable` operation on the Subscriber.

The initial value of the default SubscriberQos in the DomainParticipant are given in the following table:

Table 20 SUBSCRIBER_QOS_DEFAULT

QosPolicy	Field	Value
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	FALSE
	ordered_access	FALSE
partition	name.length	0
group_data	value.length	0
entity_factory	autoenable_ created_entities	TRUE

TopicQos

Scope

DDS

Synopsis

```
#include <ccpp_dds_dcps.h>
struct TopicQos
{
    TopicDataQosPolicy      topic_data;
    DurabilityQosPolicy     durability;
    DurabilityServiceQosPolicy durability_service;
    DeadlineQosPolicy       deadline;
    LatencyBudgetQosPolicy  latency_budget;
    LivelinessQosPolicy     liveliness;
    ReliabilityQosPolicy    reliability;
    DestinationOrderQosPolicy destination_order;
    HistoryQosPolicy        history;
    ResourceLimitsQosPolicy resource_limits;
    TransportPriorityQosPolicy transport_priority;
    LifespanQosPolicy       lifespan;
    OwnershipQosPolicy      ownership;
};
```

Description

This struct provides the basic mechanism for an application to specify Quality of Service attributes for a Topic.

Attributes

TopicDataQosPolicy topic_data - used to attach additional information to the Topic.

DurabilityQosPolicy durability - whether the data should be stored for late joining readers.

DurabilityServiceQosPolicy durability_service - the behaviour of the “transinet/persistent service” of the Data Distribution System regarding Transient and Persistent Topic instances.

DeadlineQosPolicy deadline - the period within which a new sample is expected or written.

LatencyBudgetQosPolicy latency_budget - used by the Data Distribution Service for optimization.

LivelinessQosPolicy liveliness - the way the liveliness of the Topic is asserted to the Data Distribution Service.

ReliabilityQosPolicy reliability - the reliability of the data distribution.

DestinationOrderQosPolicy destination_order - the order in which the DataReader timely orders the data.

HistoryQosPolicy history - how samples should be stored.

ResourceLimitsQosPolicy resource_limits - the maximum amount of resources to be used.

TransportPriorityQosPolicy transport_priority - a priority hint for the underlying transport layer.

LifespanQosPolicy lifespan - the maximum duration of validity of the data written by a DataWriter.

OwnershipQosPolicy ownership - whether a DataWriter exclusively owns an instance.

Detailed Description

A QosPolicy can be set when the Topic is created with the `create_topic` operation (or modified with the `set_qos` operation). Both operations take the TopicQos struct as a parameter. There may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified when they are being created and, in case they are already enabled, via the `set_qos` operation.

Some QosPolicy have “immutable” semantics meaning that they can only be specified either at Topic creation time or prior to calling the enable operation on the Topic.

The initial value of the default TopicQos in the DomainParticipant are given in the following table:

Table 21 TOPIC_QOS_DEFAULT

QosPolicy	Field	Value
topic_data	value.length	0
durability	kind	VOLATILE_DURABILITY_QOS
durability_service	service_cleanup_delay	0
	history_kind	KEEP_LAST_HISTORY_QOS
	history_depth	1
	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
deadline	period	DURATION_INFINITE
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITE
reliability	kind	BEST_EFFORT_RELIABILITY_QOS
	max_blocking_time	100 ms
	synchronous	FALSE
destination_order	kind	BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
history	kind	KEEP_LAST_HISTORY_QOS
	depth	1
resource_limits	max_samples	LENGTH_UNLIMITED
	max_instances	LENGTH_UNLIMITED
	max_samples_per_instance	LENGTH_UNLIMITED
transport_priority	value	0
lifespan	duration	DURATION_INFINITE
ownership	kind	SHARED_OWNERSHIP_QOS

B

API Constants and Types

These constants and types are taken from the `ccpp_dds_dcps.h` include file.

```

/* -----
 * Duration and Time
 * ----- */
struct Duration_t
{
    Long sec;
    ULong nanosec;
};

const Long DURATION_INFINITE_SEC      = (Long)2147483647;
const ULong DURATION_INFINITE_NSEC    = (ULong)2147483647UL;
const Long DURATION_ZERO_SEC          = (Long)0;
const ULong DURATION_ZERO_NSEC        = (ULong)0UL;

const ::DDS::Duration_t DURATION_INFINITE =
    { DURATION_INFINITE_SEC, DURATION_INFINITE_NSEC };
const ::DDS::Duration_t DURATION_ZERO =
    { 0L, 0U };

struct Time_t
{
    Long sec;
    ULong nanosec;
};

/* -----
 * Pre-defined values
 * ----- */
const LongLong HANDLE_NIL              = (LongLong)0x0;
const Long LENGTH_UNLIMITED            = (Long)-1;
const Long TIMESTAMP_INVALID_SEC       = (Long)-1;
const ULong TIMESTAMP_INVALID_NSEC     = (ULong)4294967295UL;

/*
-----
 * Return Codes
 *
-----
*/
const Long RETCODE_OK                  = (Long)0;

```

```

const Long RETCODE_ERROR = (Long)1;
const Long RETCODE_UNSUPPORTED = (Long)2;
const Long RETCODE_BAD_PARAMETER = (Long)3;
const Long RETCODE_PRECONDITION_NOT_MET = (Long)4;
const Long RETCODE_OUT_OF_RESOURCES = (Long)5;
const Long RETCODE_NOT_ENABLED = (Long)6;
const Long RETCODE_IMMUTABLE_POLICY = (Long)7;
const Long RETCODE_INCONSISTENT_POLICY = (Long)8;
const Long RETCODE_ALREADY_DELETED = (Long)9;
const Long RETCODE_TIMEOUT = (Long)10;
const Long RETCODE_NO_DATA = (Long)11;
const Long RETCODE_ILLEGAL_OPERATION = (Long)12;

/*
-----
* Status to support listeners and conditions
*
-----
*/
const ULong INCONSISTENT_TOPIC_STATUS = (ULong)1UL;
const ULong OFFERED_DEADLINE_MISSED_STATUS = (ULong)2UL;
const ULong REQUESTED_DEADLINE_MISSED_STATUS = (ULong)4UL;
const ULong OFFERED_INCOMPATIBLE_QOS_STATUS = (ULong)32UL;
const ULong REQUESTED_INCOMPATIBLE_QOS_STATUS = (ULong)64UL;
const ULong SAMPLE_LOST_STATUS = (ULong)128UL;
const ULong SAMPLE_REJECTED_STATUS = (ULong)256UL;
const ULong DATA_ON_READERS_STATUS = (ULong)512UL;
const ULong DATA_AVAILABLE_STATUS = (ULong)1024UL;
const ULong LIVELINESS_LOST_STATUS = (ULong)2048UL;
const ULong LIVELINESS_CHANGED_STATUS = (ULong)4096UL;
const ULong PUBLICATION_MATCHED_STATUS = (ULong)8192UL;
const ULong SUBSCRIPTION_MATCHED_STATUS = (ULong)16384UL;

/* Note: ANY_STATUS is deprecated, please use spec version
* specific constants.
*/
const ::DDS::StatusKind ANY_STATUS = 0x7FE7;
/* STATUS_MASK_ANY_V1_2 is all standardised status bits
* as of V1.2 of the specification.
*/
const ::DDS::StatusKind STATUS_MASK_ANY_V1_2 = 0x7FE7;
const ::DDS::StatusKind STATUS_MASK_NONE = 0x0;

/* ----- * States
* ----- */
/*
* Sample states to support reads
*/

```



```

const ULong READ_SAMPLE_STATE          = (ULong)1UL;
const ULong NOT_READ_SAMPLE_STATE      = (ULong)2UL;

/*
 * This is a bit mask SampleStateKind
 */
const ULong ANY_SAMPLE_STATE           = (ULong)65535UL;

/*
 * View states to support reads
 */
const ULong NEW_VIEW_STATE              = (ULong)1UL;
const ULong NOT_NEW_VIEW_STATE         = (ULong)2UL;

/*
 * This is a bit mask ViewStateKind
 */
const ULong ANY_SAMPLE_STATE           = (ULong)65535UL;

/*
 * Instance states to support reads
 */
const ULong ALIVE_INSTANCE_STATE        = (ULong)1UL;
const ULong NOT_ALIVE_DISPOSED_INSTANCE_STATE = (ULong)2UL;
const ULong NOT_ALIVE_NO_WRITERS_INSTANCE_STATE = (ULong)4UL;

/*
 * This is a bit mask InstanceStateKind
 */
const ULong ANY_INSTANCE_STATE          = (ULong)65535UL;
const ULong NOT_ALIVE_INSTANCE_STATE    = (ULong)6UL;

/* -----
 * Participant Factory define
 * ----- */
#define TheParticipantFactory
    (::DDS::DomainParticipantFactory::get_instance())

/* -----
 * Qos defines
 * ----- */
#define TheParticipantFactory
    (::DDS::DomainParticipantFactory::get_instance())
#define PARTICIPANT_QOS_DEFAULT
    (*::DDS::DomainParticipantFactory::participant_qos_default())
#define TOPIC_QOS_DEFAULT
    (*::DDS::DomainParticipantFactory::topic_qos_default())

```

```

#define PUBLISHER_QOS_DEFAULT
    (*::DDS::DomainParticipantFactory::publisher_qos_default())
#define SUBSCRIBER_QOS_DEFAULT
    (*::DDS::DomainParticipantFactory::subscriber_qos_default())
#define DATAREADER_QOS_DEFAULT
    (*::DDS::DomainParticipantFactory::datareader_qos_default())
#define DATAREADER_QOS_USE_TOPIC_QOS
    (*::DDS::DomainParticipantFactory::datareader_qos_use_topic_qos())
#define DATAWRITER_QOS_DEFAULT
    (*::DDS::DomainParticipantFactory::datawriter_qos_default())
#define DATAWRITER_QOS_USE_TOPIC_QOS
    (*::DDS::DomainParticipantFactory::datawriter_qos_use_topic_qos())

/* -----
 * QosPolicy
 * ----- */
const String USERDATA_QOS_POLICY_NAME =
    (String)"UserData";
const String DURABILITY_QOS_POLICY_NAME =
    (String)"Durability";
const String PRESENTATION_QOS_POLICY_NAME =
    (String)"Presentation";
const String DEADLINE_QOS_POLICY_NAME =
    (String)"Deadline";
const String LATENCYBUDGET_QOS_POLICY_NAME =
    (String)"LatencyBudget";
const String OWNERSHIP_QOS_POLICY_NAME =
    (String)"Ownership";
const String OWNERSHIPSTRENGTH_QOS_POLICY_NAME =
    (String)"OwnershipStrength";
const String LIVELINESS_QOS_POLICY_NAME =
    (String)"Liveliness";
const String TIMEBASEDFILTER_QOS_POLICY_NAME =
    (String)"TimeBasedFilter";
const String PARTITION_QOS_POLICY_NAME =
    (String)"Partition";
const String RELIABILITY_QOS_POLICY_NAME =
    (String)"Reliability";
const String DESTINATIONORDER_QOS_POLICY_NAME =
    (String)"DestinationOrder";
const String HISTORY_QOS_POLICY_NAME =
    (String)"History";
const String RESOURCELIMITS_QOS_POLICY_NAME =
    (String)"ResourceLimits";
const String ENTITYFACTORY_QOS_POLICY_NAME =
    (String)"EntityFactory";
const String WRITERDATALIFECYCLE_QOS_POLICY_NAME =
    (String)"WriterDataLifecycle";
const String READERDATALIFECYCLE_QOS_POLICY_NAME =

```

```

        (String) "ReaderDataLifecycle";
const String TOPICDATA_QOS_POLICY_NAME          =
        (String) "TopicData";
const String GROUPDATA_QOS_POLICY_NAME          =
        (String) "GroupData";
const String TRANSPORTPRIORITY_QOS_POLICY_NAME  =
        (String) "TransportPriority";
const String LIFESPAN_QOS_POLICY_NAME           =
        (String) "Lifespan";
const String DURABILITYSERVICE_QOS_POLICY_NAME =
        (String) "DurabilityService";

const Long INVALID_QOS_POLICY_ID                = (Long) 0;
const Long USERDATA_QOS_POLICY_ID              = (Long) 1;
const Long DURABILITY_QOS_POLICY_ID            = (Long) 2;
const Long PRESENTATION_QOS_POLICY_ID          = (Long) 3;
const Long DEADLINE_QOS_POLICY_ID              = (Long) 4;
const Long LATENCYBUDGET_QOS_POLICY_ID        = (Long) 5;
const Long OWNERSHIP_QOS_POLICY_ID             = (Long) 6;
const Long OWNERSHIPSTRENGTH_QOS_POLICY_ID     = (Long) 7;
const Long LIVELINESS_QOS_POLICY_ID            = (Long) 8;
const Long TIMEBASEDFILTER_QOS_POLICY_ID       = (Long) 9;
const Long PARTITION_QOS_POLICY_ID             = (Long) 10;
const Long RELIABILITY_QOS_POLICY_ID           = (Long) 11;
const Long DESTINATIONORDER_QOS_POLICY_ID      = (Long) 12;
const Long HISTORY_QOS_POLICY_ID               = (Long) 13;
const Long RESOURCELIMITS_QOS_POLICY_ID        = (Long) 14;
const Long ENTITYFACTORY_QOS_POLICY_ID         = (Long) 15;
const Long WRITERDATA_LIFECYCLE_QOS_POLICY_ID  = (Long) 16;
const Long READERDATA_LIFECYCLE_QOS_POLICY_ID  = (Long) 17;
const Long TOPICDATA_QOS_POLICY_ID            = (Long) 18;
const Long GROUPDATA_QOS_POLICY_ID            = (Long) 19;
const Long TRANSPORTPRIORITY_QOS_POLICY_ID     = (Long) 20;
const Long LIFESPAN_QOS_POLICY_ID              = (Long) 21;
const Long DURABILITYSERVICE_QOS_POLICY_ID    = (Long) 22;

```


C Platform Specific Model IDL Interface

The IDL code in the next paragraphs are taken from the *OMG C++ Language Mapping Specification*.

dds_dcps.idl

```
#define DOMAINID_TYPE_NATIVE string
#define HANDLE_TYPE_NATIVE long long
#define HANDLE_NIL_NATIVE 0
#define BUILTIN_TOPIC_KEY_TYPE_NATIVE long
#define TheParticipantFactory
#define PARTICIPANT_QOS_DEFAULT
#define TOPIC_QOS_DEFAULT
#define PUBLISHER_QOS_DEFAULT
#define SUBSCRIBER_QOS_DEFAULT
#define DATAWRITER_QOS_DEFAULT
#define DATAREADER_QOS_DEFAULT
#define DATAWRITER_QOS_USE_TOPIC_QOS
#define DATAREADER_QOS_USE_TOPIC_QOS
module DDS {
    typedef DOMAINID_TYPE_NATIVE DomainId_t;
    typedef HANDLE_TYPE_NATIVE InstanceHandle_t;
    typedef BUILTIN_TOPIC_KEY_TYPE_NATIVE BuiltinTopicKey_t[3];
    typedef sequence<InstanceHandle_t> InstanceHandleSeq;
    typedef long ReturnCode_t;
    typedef long QosPolicyId_t;
    typedef sequence<string> StringSeq;
    struct Duration_t {
        long sec;
        unsigned long nanosec;
    };
    struct Time_t {
        long sec;
        unsigned long nanosec;
    };
    //
    // Pre-defined values
    //
    const InstanceHandle_t HANDLE_NIL = HANDLE_NIL_NATIVE;
    const long LENGTH_UNLIMITED = -1;
    const long DURATION_INFINITE_SEC = 0x7fffffff;
```

```

const unsigned long DURATION_INFINITE_NSEC= 0x7fffffff;
const long DURATION_ZERO_SEC= 0;
const unsigned long DURATION_ZERO_NSEC= 0;
const long TIMESTAMP_INVALID_SEC= -1;
const unsigned long TIMESTAMP_INVALID_NSEC= 0xffffffff;
//
// Return codes
//
const ReturnCode_t RETCODE_OK = 0;
const ReturnCode_t RETCODE_ERROR = 1;
const ReturnCode_t RETCODE_UNSUPPORTED = 2;
const ReturnCode_t RETCODE_BAD_PARAMETER = 3;
const ReturnCode_t RETCODE_PRECONDITION_NOT_MET = 4;
const ReturnCode_t RETCODE_OUT_OF_RESOURCES = 5;
const ReturnCode_t RETCODE_NOT_ENABLED = 6;
const ReturnCode_t RETCODE_IMMUTABLE_POLICY = 7;
const ReturnCode_t RETCODE_INCONSISTENT_POLICY = 8;
const ReturnCode_t RETCODE_ALREADY_DELETED = 9;
const ReturnCode_t RETCODE_TIMEOUT = 10;
const ReturnCode_t RETCODE_NO_DATA = 11;
const ReturnCode_t RETCODE_ILLEGAL_OPERATION = 12;

//
// Status to support listeners and conditions
//
typedef unsigned long StatusKind;
typedef unsigned long StatusMask; // bit-mask StatusKind
const StatusKind INCONSISTENT_TOPIC_STATUS = 0x0001 << 0;
const StatusKind OFFERED_DEADLINE_MISSED_STATUS = 0x0001 << 1;
const StatusKind REQUESTED_DEADLINE_MISSED_STATUS = 0x0001 << 2;
const StatusKind OFFERED_INCOMPATIBLE_QOS_STATUS = 0x0001 << 5;
const StatusKind REQUESTED_INCOMPATIBLE_QOS_STATUS= 0x0001 << 6;
const StatusKind SAMPLE_LOST_STATUS = 0x0001 << 7;
const StatusKind SAMPLE_REJECTED_STATUS = 0x0001 << 8;
const StatusKind DATA_ON_READERS_STATUS = 0x0001 << 9;
const StatusKind DATA_AVAILABLE_STATUS = 0x0001 << 10;
const StatusKind LIVELINESS_LOST_STATUS = 0x0001 << 11;
const StatusKind LIVELINESS_CHANGED_STATUS = 0x0001 << 12;
const StatusKind PUBLICATION_MATCHED_STATUS = 0x0001 << 13;
const StatusKind SUBSCRIPTION_MATCHED_STATUS = 0x0001 << 14;
struct InconsistentTopicStatus {
    long total_count;
    long total_count_change;
};
struct SampleLostStatus {
    long total_count;
    long total_count_change;
};
enum SampleRejectedStatusKind {
    NOT_REJECTED,

```

```

    REJECTED_BY_INSTANCE_LIMIT,
    REJECTED_BY_SAMPLES_LIMIT,
    REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
};
struct SampleRejectedStatus {
    long total_count;
    long total_count_change;
    SampleRejectedStatusKind last_reason;
    InstanceHandle_t last_instance_handle;
};
struct LivelinessLostStatus {
    long total_count;
    long total_count_change;
};
struct LivelinessChangedStatus {
    long alive_count;
    long not_alive_count;
    long alive_count_change;
    long not_alive_count_change;
    InstanceHandle_t last_publication_handle;
};
struct OfferedDeadlineMissedStatus {
    long total_count;
    long total_count_change;
    InstanceHandle_t last_instance_handle;
};
struct RequestedDeadlineMissedStatus {
    long total_count;
    long total_count_change;
    InstanceHandle_t last_instance_handle;
};
struct QosPolicyCount {
    QosPolicyId_t policy_id;
    long count;
};
typedef sequence<QosPolicyCount> QosPolicyCountSeq;
struct OfferedIncompatibleQosStatus {
    long total_count;
    long total_count_change;
    QosPolicyId_t last_policy_id;
    QosPolicyCountSeq policies;
};
struct RequestedIncompatibleQosStatus {
    long total_count;
    long total_count_change;
    QosPolicyId_t last_policy_id;
    QosPolicyCountSeq policies;
};
struct PublicationMatchedException {
    long total_count;

```

```

        long total_count_change;
        long current_count;
        long current_count_change;
        InstanceHandle_t last_subscription_handle;
};
struct SubscriptionMatchedStatus {
    long total_count;
    long total_count_change;
    long current_count;
    long current_count_change;
    InstanceHandle_t last_publication_handle;
};
//
// Listeners
//
interface Listener;
interface Entity;
interface TopicDescription;
interface Topic;
interface ContentFilteredTopic;
interface MultiTopic;
interface DataWriter;
interface DataReader;
interface Subscriber;
interface Publisher;
typedef sequence<Topic> TopicSeq;
typedef sequence<DataReader> DataReaderSeq;
interface Listener {
};
interface TopicListener : Listener {
void
    on_inconsistent_topic(
        in Topic the_topic,
        in InconsistentTopicStatus status);
};
interface DataWriterListener : Listener {
void
    on_offered_deadline_missed(
        in DataWriter writer,
        in OfferedDeadlineMissedStatus status);
void
    on_offered_incompatible_qos(
        in DataWriter writer,
        in OfferedIncompatibleQosStatus status);
void
    on_liveliness_lost(
        in DataWriter writer,
        in LivelinessLostStatus status);
void
    on_publication_matched(

```



```

        in DataWriter writer,
        in PublicationMatchedStatus status);
};
interface PublisherListener : DataWriterListener {
};
interface DataReaderListener : Listener {
void
    on_requested_deadline_missed(
        in DataReader reader,
        in RequestedDeadlineMissedStatus status);
void
    on_requested_incompatible_qos(
        in DataReader reader,
        in RequestedIncompatibleQosStatus status);
void
    on_sample_rejected(
        in DataReader reader,
        in SampleRejectedStatus status);
void
    on_liveliness_changed(
        in DataReader reader,
        in LivelinessChangedStatus status);
void
    on_data_available(
        in DataReader reader);
void
    on_subscription_matched(
        in DataReader reader,
        in SubscriptionMatchedStatus status);
void
    on_sample_lost(
        in DataReader reader,
        in SampleLostStatus status);
};
interface SubscriberListener : DataReaderListener {
void
    on_data_on_readers(
        in Subscriber subs);
};
interface DomainParticipantListener : TopicListener,
    PublisherListener,
    SubscriberListener {
};
//
// Conditions
//
interface Condition {
boolean
get_trigger_value();
};

```

```

typedef sequence<Condition> ConditionSeq;
interface WaitSet {
ReturnCode_t
wait(
    inout ConditionSeq active_conditions,
    in Duration_t timeout);
ReturnCode_t
attach_condition(
    in Condition cond);
ReturnCode_t
detach_condition(
    in Condition cond);
ReturnCode_t
get_conditions(
    inout ConditionSeq attached_conditions);
};
interface GuardCondition : Condition {
ReturnCode_t
set_trigger_value(
    in boolean value);
};
interface StatusCondition : Condition {
StatusMask
get_enabled_statuses();
ReturnCode_t
set_enabled_statuses(
    in StatusMask mask);
Entity
get_entity();
};
// Sample states to support reads
typedef unsigned long SampleStateKind;
typedef sequence <SampleStateKind> SampleStateSeq;
const SampleStateKind READ_SAMPLE_STATE = 0x0001 << 0;
const SampleStateKind NOT_READ_SAMPLE_STATE = 0x0001 << 1;
// This is a bit-mask SampleStateKind
typedef unsigned long SampleStateMask;
const SampleStateMask ANY_SAMPLE_STATE = 0xffff;
// View states to support reads
typedef unsigned long ViewStateKind;
typedef sequence<ViewStateKind> ViewStateSeq;
const ViewStateKind NEW_VIEW_STATE = 0x0001 << 0;
const ViewStateKind NOT_NEW_VIEW_STATE = 0x0001 << 1;
// This is a bit-mask ViewStateKind
typedef unsigned long ViewStateMask;
const ViewStateMask ANY_VIEW_STATE = 0xffff;
// Instance states to support reads
typedef unsigned long InstanceStateKind;
typedef sequence<InstanceStateKind> InstanceStateSeq;
const InstanceStateKind ALIVE_INSTANCE_STATE = 0x0001 << 0;

```

```

const InstanceStateKind NOT_ALIVE_DISPOSED_INSTANCE_STATE = 0x0001
    << 1;
const InstanceStateKind NOT_ALIVE_NO_WRITERS_INSTANCE_STATE =
    0x0001 << 2;
// This is a bit-mask InstanceStateKind
typedef unsigned long InstanceStateMask;
const InstanceStateMask ANY_INSTANCE_STATE = 0xffff;
const InstanceStateMask NOT_ALIVE_INSTANCE_STATE = 0x006;
interface ReadCondition : Condition {
SampleStateMask
get_sample_state_mask();
ViewStateMask
get_view_state_mask();
InstanceStateMask
get_instance_state_mask();
DataReader
get_datareader();
};
interface QueryCondition : ReadCondition {
string
get_query_expression();
ReturnCode_t
get_query_parameters(
    inout StringSeq query_parameters);
ReturnCode_t
set_query_parameters(
    in StringSeq query_parameters);
};
//
// Qos
//
const string USERDATA_QOS_POLICY_NAME          = "UserData";
const string DURABILITY_QOS_POLICY_NAME         = "Durability";
const string PRESENTATION_QOS_POLICY_NAME       = "Presentation";
const string DEADLINE_QOS_POLICY_NAME          = "Deadline";
const string LATENCYBUDGET_QOS_POLICY_NAME      = "LatencyBudget";
const string OWNERSHIP_QOS_POLICY_NAME          = "Ownership";
const string OWNERSHIPSTRENGTH_QOS_POLICY_NAME=
    "OwnershipStrength";
const string LIVELINESS_QOS_POLICY_NAME         = "Liveliness";
const string TIMEBASEDFILTER_QOS_POLICY_NAME= "TimeBasedFilter";
const string PARTITION_QOS_POLICY_NAME         = "Partition";
const string RELIABILITY_QOS_POLICY_NAME       = "Reliability";
const string DESTINATIONORDER_QOS_POLICY_NAME =
    "DestinationOrder";
const string HISTORY_QOS_POLICY_NAME           = "History";
const string RESOURCELIMITS_QOS_POLICY_NAME= "ResourceLimits";
const string ENTITYFACTORY_QOS_POLICY_NAME     = "EntityFactory";
const string WRITERDATA LifecYCLE_QOS_POLICY_NAM=
    "WriterDataLifecYcle";

```

```

const string READERDATA_LIFECYCLE_QOS_POLICY_NAME=
    "ReaderDataLifecycle";
const string TOPICDATA_QOS_POLICY_NAME          = "TopicData";
const string GROUPDATA_QOS_POLICY_NAME          = "GroupData";
const string TRANSPORTPRIORITY_QOS_POLICY_NAME=
    "TransportPriority";
const string LIFESPAN_QOS_POLICY_NAME           = "Lifespan";
const string DURABILITYSERVICE_QOS_POLICY_NAME=
    "DurabilityService";
const QosPolicyId_t INVALID_QOS_POLICY_ID       = 0;
const QosPolicyId_t USERDATA_QOS_POLICY_ID     = 1;
const QosPolicyId_t DURABILITY_QOS_POLICY_ID    = 2;
const QosPolicyId_t PRESENTATION_QOS_POLICY_ID  = 3;
const QosPolicyId_t DEADLINE_QOS_POLICY_ID      = 4;
const QosPolicyId_t LATENCYBUDGET_QOS_POLICY_ID = 5;
const QosPolicyId_t OWNERSHIP_QOS_POLICY_ID      = 6;
const QosPolicyId_t OWNERSHIPSTRENGTH_QOS_POLICY_ID = 7;
const QosPolicyId_t LIVELINESS_QOS_POLICY_ID    = 8;
const QosPolicyId_t TIMEBASEDFILTER_QOS_POLICY_ID = 9;
const QosPolicyId_t PARTITION_QOS_POLICY_ID     = 10;
const QosPolicyId_t RELIABILITY_QOS_POLICY_ID   = 11;
const QosPolicyId_t DESTINATIONORDER_QOS_POLICY_ID = 12;
const QosPolicyId_t HISTORY_QOS_POLICY_ID       = 13;
const QosPolicyId_t RESOURCELIMITS_QOS_POLICY_ID = 14;
const QosPolicyId_t ENTITYFACTORY_QOS_POLICY_ID = 15;
const QosPolicyId_t WRITERDATA_LIFECYCLE_QOS_POLICY_ID= 16;
const QosPolicyId_t READERDATA_LIFECYCLE_QOS_POLICY_ID= 17;
const QosPolicyId_t TOPICDATA_QOS_POLICY_ID     = 18;
const QosPolicyId_t GROUPDATA_QOS_POLICY_ID     = 19;
const QosPolicyId_t TRANSPORTPRIORITY_QOS_POLICY_ID = 20;
const QosPolicyId_t LIFESPAN_QOS_POLICY_ID      = 21;
const QosPolicyId_t DURABILITYSERVICE_QOS_POLICY_ID = 22;
struct UserDataQosPolicy {
sequence<octet> value;
};
struct TopicDataQosPolicy {
sequence<octet> value;
};
struct GroupDataQosPolicy {
sequence<octet> value;
};
struct TransportPriorityQosPolicy {
long value;
};
struct LifespanQosPolicy {
Duration_t duration;
};
enum DurabilityQosPolicyKind {
VOLATILE_DURABILITY_QOS,
TRANSIENT_LOCAL_DURABILITY_QOS,

```

```

TRANSIENT_DURABILITY_QOS,
PERSISTENT_DURABILITY_QOS
};
struct DurabilityQosPolicy {
DurabilityQosPolicyKind kind;
};
enum PresentationQosPolicyAccessScopeKind {
INSTANCE_PRESENTATION_QOS,
TOPIC_PRESENTATION_QOS,
GROUP_PRESENTATION_QOS
};
struct PresentationQosPolicy {
    PresentationQosPolicyAccessScopeKind access_scope;
    boolean coherent_access;
    boolean ordered_access;
};
struct DeadlineQosPolicy {
    Duration_t period;
};
struct LatencyBudgetQosPolicy {
    Duration_t duration;
};
enum OwnershipQosPolicyKind {
    SHARED_OWNERSHIP_QOS,
    EXCLUSIVE_OWNERSHIP_QOS
};
struct OwnershipQosPolicy {
    OwnershipQosPolicyKind kind;
};
struct OwnershipStrengthQosPolicy {
    long value;
};
enum LivelinessQosPolicyKind {
    AUTOMATIC_LIVELINESS_QOS,
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
    MANUAL_BY_TOPIC_LIVELINESS_QOS
};
struct LivelinessQosPolicy {
    LivelinessQosPolicyKind kind;
    Duration_t lease_duration;
};
struct TimeBasedFilterQosPolicy {
    Duration_t minimum_separation;
};
struct PartitionQosPolicy {
    StringSeq name;
};
enum ReliabilityQosPolicyKind {
    BEST_EFFORT_RELIABILITY_QOS,
    RELIABLE_RELIABILITY_QOS
};

```

```

};
struct ReliabilityQosPolicy {
    ReliabilityQosPolicyKind kind;
    Duration_t max_blocking_time;
    boolean synchronous;
};
enum DestinationOrderQosPolicyKind {
    BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
    BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
};
struct DestinationOrderQosPolicy {
    DestinationOrderQosPolicyKind kind;
};
enum HistoryQosPolicyKind {
    KEEP_LAST_HISTORY_QOS,
    KEEP_ALL_HISTORY_QOS
};
struct HistoryQosPolicy {
    HistoryQosPolicyKind kind;
    long depth;
};
struct ResourceLimitsQosPolicy {
    long max_samples;
    long max_instances;
    long max_samples_per_instance;
};
struct EntityFactoryQosPolicy {
    boolean autoenable_created_entities;
};
struct WriterDataLifecycleQosPolicy {
    boolean autodispose_unregistered_instances;
};
struct ReaderDataLifecycleQosPolicy {
    Duration_t autopurge_nowriter_samples_delay;
    Duration_t autopurge_disposed_samples_delay;
};
struct DurabilityServiceQosPolicy {
    Duration_t service_cleanup_delay;
    HistoryQosPolicyKind history_kind;
    long history_depth;
    long max_samples;
    long max_instances;
    long max_samples_per_instance;
};
struct DomainParticipantFactoryQos {
    EntityFactoryQosPolicy entity_factory;
};
struct DomainParticipantQos {
    UserDataQosPolicy user_data;
    EntityFactoryQosPolicy entity_factory;
};

```

```

};
struct TopicQos {
    TopicDataQosPolicy topic_data;
    DurabilityQosPolicy durability;
    DurabilityServiceQosPolicy durability_service;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
    ReliabilityQosPolicy reliability;
    DestinationOrderQosPolicy destination_order;
    HistoryQosPolicy history;
    ResourceLimitsQosPolicy resource_limits;
    TransportPriorityQosPolicy transport_priority;
    LifespanQosPolicy lifespan;
    OwnershipQosPolicy ownership;
};
struct DataWriterQos {
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
    ReliabilityQosPolicy reliability;
    DestinationOrderQosPolicy destination_order;
    HistoryQosPolicy history;
    ResourceLimitsQosPolicy resource_limits;
    TransportPriorityQosPolicy transport_priority;
    LifespanQosPolicy lifespan;
    UserDataQosPolicy user_data;
    OwnershipQosPolicy ownership;
    OwnershipStrengthQosPolicy ownership_strength;
    WriterDataLifecycleQosPolicy writer_data_lifecycle;
};
struct PublisherQos {
    PresentationQosPolicy presentation;
    PartitionQosPolicy partition;
    GroupDataQosPolicy group_data;
    EntityFactoryQosPolicy entity_factory;
};
struct DataReaderQos {
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
    ReliabilityQosPolicy reliability;
    DestinationOrderQosPolicy destination_order;
    HistoryQosPolicy history;
    ResourceLimitsQosPolicy resource_limits;
    UserDataQosPolicy user_data;
    OwnershipQosPolicy ownership;
    TimeBasedFilterQosPolicy time_based_filter;
};

```

```

        ReaderDataLifecycleQosPolicy reader_data_lifecycle;
    };
    struct SubscriberQos {
        PresentationQosPolicy presentation;
        PartitionQosPolicy partition;
        GroupDataQosPolicy group_data;
        EntityFactoryQosPolicy entity_factory;
    };
    //
    struct ParticipantBuiltinTopicData {
        BuiltinTopicKey_t key;
        UserDataQosPolicy user_data;
    };
    struct TopicBuiltinTopicData {
        BuiltinTopicKey_t key;
        string name;
        string type_name;
        DurabilityQosPolicy durability;
        DeadlineQosPolicy deadline;
        LatencyBudgetQosPolicy latency_budget;
        LivelinessQosPolicy liveliness;
        ReliabilityQosPolicy reliability;
        TransportPriorityQosPolicy transport_priority;
        LifespanQosPolicy lifespan;
        DestinationOrderQosPolicy destination_order;
        HistoryQosPolicy history;
        ResourceLimitsQosPolicy resource_limits;
        OwnershipQosPolicy ownership;
        TopicDataQosPolicy topic_data;
    };
    struct PublicationBuiltinTopicData {
        BuiltinTopicKey_t key;
        BuiltinTopicKey_t participant_key;
        string topic_name;
        string type_name;
        DurabilityQosPolicy durability;
        DeadlineQosPolicy deadline;
        LatencyBudgetQosPolicy latency_budget;
        LivelinessQosPolicy liveliness;
        ReliabilityQosPolicy reliability;
        LifespanQosPolicy lifespan;
        UserDataQosPolicy user_data;
        OwnershipStrengthQosPolicy ownership_strength;
        PresentationQosPolicy presentation;
        PartitionQosPolicy partition;
        TopicDataQosPolicy topic_data;
        GroupDataQosPolicy group_data;
    };
    struct SubscriptionBuiltinTopicData {
        BuiltinTopicKey_t key;

```



```

    BuiltinTopicKey_t participant_key;
    string topic_name;
    string type_name;
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
    ReliabilityQosPolicy reliability;
    DestinationOrderQosPolicy destination_order;
    UserDataQosPolicy user_data;
    TimeBasedFilterQosPolicy time_based_filter;
    PresentationQosPolicy presentation;
    PartitionQosPolicy partition;
    TopicDataQosPolicy topic_data;
    GroupDataQosPolicy group_data;
};
//
interface Entity {
//   ReturnCode_t
//   set_qos(
//       in EntityQos qos);
//
//   ReturnCode_t
//   get_qos(
//       inout EntityQos qos);
//
//   ReturnCode_t
//   set_listener(
//       in Listener l,
//       in StatusMask mask);
//
//   Listener
//   get_listener();
ReturnCode_t
enable();
StatusCondition
get_statuscondition();
StatusMask
get_status_changes();
};
//
interface DomainParticipant : Entity {
    // Factory interfaces
    Publisher
    create_publisher(
        in PublisherQos qos,
        in PublisherListener a_listener,
        in StatusMask mask);
ReturnCode_t
delete_publisher(

```

```

        in Publisher p);
Subscriber
create_subscriber(
    in SubscriberQos qos,
    in SubscriberListener a_listener,
    in StatusMask mask);
ReturnCode_t
delete_subscriber(
    in Subscriber s);
Subscriber
get_builtin_subscriber();
Topic
create_topic(
    in string topic_name,
    in string type_name,
    in TopicQos qos,
    in TopicListener a_listener,
    in StatusMask mask);
ReturnCode_t
delete_topic(
    in Topic a_topic);
Topic
find_topic(
    in string topic_name,
    in Duration_t timeout);
TopicDescription
lookup_topicdescription(
    in string name);
ContentFilteredTopic
create_contentfilteredtopic(
    in string name,
    in Topic related_topic,
    in string filter_expression,
    in StringSeq expression_parameters);
ReturnCode_t
delete_contentfilteredtopic(
    in ContentFilteredTopic a_contentfilteredtopic);
MultiTopic
create_multitopic(
    in string name,
    in string type_name,
    in string subscription_expression,
    in StringSeq expression_parameters);
ReturnCode_t
delete_multitopic(
    in MultiTopic a_multitopic);
ReturnCode_t
delete_contained_entities();
ReturnCode_t
set_qos(

```

```

        in DomainParticipantQos qos);
ReturnCode_t
get_qos(
    inout DomainParticipantQos qos);
ReturnCode_t
set_listener(
    in DomainParticipantListener a_listener,
    in StatusMask mask);
DomainParticipantListener
get_listener();
ReturnCode_t
ignore_participant(
    in InstanceHandle_t handle);
ReturnCode_t
ignore_topic(
    in InstanceHandle_t handle);
ReturnCode_t
ignore_publication(
    in InstanceHandle_t handle);
ReturnCode_t
ignore_subscription(
    in InstanceHandle_t handle);
DomainId_t
get_domain_id();
ReturnCode_t
assert_liveliness();
ReturnCode_t
set_default_publisher_qos(
    in PublisherQos qos);
ReturnCode_t
get_default_publisher_qos(
    inout PublisherQos qos);
ReturnCode_t
set_default_subscriber_qos(
    in SubscriberQos qos);
ReturnCode_t
get_default_subscriber_qos(
    inout SubscriberQos qos);
ReturnCode_t
set_default_topic_qos(
    in TopicQos qos);
ReturnCode_t
get_default_topic_qos(
    inout TopicQos qos);
boolean
contains_entity(
    in InstanceHandle_t a_handle);
ReturnCode_t
get_current_time(
    inout Time_t current_time);

```

```

};
interface DomainParticipantFactory {
//
//  DomainParticipantFactory
//  get_instance();
//
DomainParticipant
create_participant(
    in DomainId_t domainId,
    in DomainParticipantQos qos,
    in DomainParticipantListener a_listener,
    in StatusMask mask);
ReturnCode_t
delete_participant(
    in DomainParticipant a_participant);
DomainParticipant
lookup_participant(
    in DomainId_t domainId);
ReturnCode_t
set_default_participant_qos(
    in DomainParticipantQos qos);
ReturnCode_t
get_default_participant_qos(
    inout DomainParticipantQos qos);

ReturnCode_t
set_qos(
    in DomainParticipantFactoryQos qos);
ReturnCode_t
get_qos(
    inout DomainParticipantFactoryQos qos);
ReturnCode_t
delete_domain
    (in Domain a_domain);
Domain
lookup_domain
    (in DomainId_t domainId);
ReturnCode_t
    create_persistent_snapshot(
        in string partition_expression,
        in string topic_expression,
        in string URI);
};
interface TypeSupport {
//  ReturnCode_t
//  register_type(
//      in DomainParticipant domain,
//      in string type_name);
//
//  string

```

```

//  get_type_name();
};
//
interface TopicDescription {
string
    get_type_name();
string
    get_name();
DomainParticipant
    get_participant();
};
interface Topic : Entity, TopicDescription {
ReturnCode_t
set_qos(
    in TopicQos qos);
ReturnCode_t
get_qos(
    inout TopicQos qos);
ReturnCode_t
set_listener(
    in TopicListener a_listener,
    in StatusMask mask);
TopicListener_ptr
get_listener();
// Access the status
ReturnCode_t
get_inconsistent_topic_status(
    inout InconsistentTopicStatus a_status);
};
interface ContentFilteredTopic : TopicDescription {
string
get_filter_expression();
ReturnCode_t
get_expression_parameters(
    inout StringSeq expression_parameters);
ReturnCode_t
set_expression_parameters(
    in StringSeq expression_parameters);
Topic
get_related_topic();
};
interface MultiTopic : TopicDescription {
string
get_subscription_expression();
ReturnCode_t
get_expression_parameters(
    inout StringSeq expression_parameters);
ReturnCode_t
set_expression_parameters(
    in StringSeq expression_parameters);
};

```

```

};
//
interface Publisher : Entity {
  DataWriter
  create_datawriter(
    in Topic a_topic,
    in DataWriterQos qos,
    in DataWriterListener a_listener,
    in StatusMask mask);
  ReturnCode_t
  delete_datawriter(
    in DataWriter a_datawriter);
  DataWriter
  lookup_datawriter(
    in string topic_name);
  ReturnCode_t
  delete_contained_entities();
  ReturnCode_t
  set_qos(
    in PublisherQos qos);
  ReturnCode_t
  get_qos(
    inout PublisherQos qos);
  ReturnCode_t
  set_listener(
    in PublisherListener a_listener,
    in StatusMask mask);
  PublisherListener
  get_listener();
  ReturnCode_t
  suspend_publications();
  ReturnCode_t
  resume_publications();
  ReturnCode_t
  begin_coherent_changes();
  ReturnCode_t
  end_coherent_changes();
  ReturnCode_t
  wait_for_acknowledgments(
    in Duration_t max_wait);
  DomainParticipant
  get_participant();
  ReturnCode_t
  set_default_datawriter_qos(
    in DataWriterQos qos);
  ReturnCode_t
  get_default_datawriter_qos(
    inout DataWriterQos qos);
  ReturnCode_t
  copy_from_topic_qos(

```

```

        inout DataWriterQos a_datawriter_qos,
        in TopicQos a_topic_qos);
};
interface DataWriter : Entity {
//  InstanceHandle_t
//  register_instance(
//      in Data instance_data);
//
//  InstanceHandle_t
//  register_instance_w_timestamp(
//      in Data instance_data,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  unregister_instance(
//      in Data instance_data,
//      in InstanceHandle_t handle);
//
//  ReturnCode_t
//  unregister_instance_w_timestamp(
//      in Data instance_data,
//      in InstanceHandle_t handle,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  write(
//      in Data instance_data,
//      in InstanceHandle_t handle);
//
//  ReturnCode_t
//  write_w_timestamp(
//      in Data instance_data,
//      in InstanceHandle_t handle,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  dispose(
//      in Data instance_data,
//      in InstanceHandle_t instance_handle);
//
//  ReturnCode_t
//  dispose_w_timestamp(
//      in Data instance_data,
//      in InstanceHandle_t instance_handle,
//      in Time_t source_timestamp);
//
//  ReturnCode_t
//  get_key_value(
//      inout Data key_holder,
//      in InstanceHandle_t handle);

```

```

//
// InstanceHandle_t lookup_instance(
//     in Data instance_data);
ReturnCode_t
set_qos(
    in DataWriterQos qos);
ReturnCode_t
get_qos(
    inout DataWriterQos qos);
ReturnCode_t
set_listener(
    in DataWriterListener a_listener,
    in StatusMask mask);
DataWriterListener
get_listener();
Topic
get_topic();
Publisher
get_publisher();
ReturnCode_t
wait_for_acknowledgments(
    in Duration_t max_wait);
// Access the status
ReturnCode_t
get_liveliness_lost_status(
    inout LivelinessLostStatus status);
ReturnCode_t
get_offered_deadline_missed_status(
    inout OfferedDeadlineMissedStatus status);
ReturnCode_t
get_offered_incompatible_qos_status(
    inout OfferedIncompatibleQosStatus status);
ReturnCode_t
get_publication_matched_status(
    inout PublicationMatchedStatus status);
ReturnCode_t
    assert_liveliness();
ReturnCode_t
    get_matched_subscriptions(
        inout InstanceHandleSeq subscription_handles);
ReturnCode_t
    get_matched_subscription_data(
        inout SubscriptionBuiltinTopicData subscription_data,
        in InstanceHandle_t subscription_handle);
};
//
interface Subscriber : Entity {
DataReader
create_datareader(
    in TopicDescription a_topic,

```



```

        in DataReaderQos qos,
        in DataReaderListener a_listener,
        in StatusMask mask);
ReturnCode_t
delete_datareader(
    in DataReader a_datareader);
ReturnCode_t
delete_contained_entities();
DataReader
lookup_datareader(
    in string topic_name);
ReturnCode_t
get_datareaders(
    inout DataReaderSeq readers,
    in SampleStateMask sample_states,
    in ViewStateMask view_states,
    in InstanceStateMask instance_states);
ReturnCode_t
notify_datareaders();
ReturnCode_t
set_qos(
    in SubscriberQos qos);
ReturnCode_t
get_qos(
    inout SubscriberQos qos);
ReturnCode_t
set_listener(
    in SubscriberListener a_listener,
    in StatusMask mask);
SubscriberListener
get_listener();
ReturnCode_t
begin_access();
ReturnCode_t
end_access();
DomainParticipant
get_participant();
ReturnCode_t
set_default_datareader_qos(
    in DataReaderQos qos);
ReturnCode_t
get_default_datareader_qos(
    inout DataReaderQos qos);
ReturnCode_t
copy_from_topic_qos(
    inout DataReaderQos a_datareader_qos,
    in TopicQos a_topic_qos);
};
interface DataReader : Entity {
    // ReturnCode_t

```

```

// read(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in SampleStateMask sample_states,
//     in ViewStateMask view_states,
//     in InstanceStateMask instance_states);
//
// ReturnCode_t
// take(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in SampleStateMask sample_states,
//     in ViewStateMask view_states,
//     in InstanceStateMask instance_states);
//
// ReturnCode_t
// read_w_condition(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in ReadCondition a_condition);
//
// ReturnCode_t
// take_w_condition(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in ReadCondition a_condition);
//
// ReturnCode_t
// read_next_sample(
//     inout Data data_values,
//     inout SampleInfo sample_info);
//
// ReturnCode_t
// take_next_sample(
//     inout Data data_values,
//     inout SampleInfo sample_info);
//
// ReturnCode_t
// read_instance(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in InstanceHandle_t a_handle,
//     in SampleStateMask sample_states,
//     in ViewStateMask view_states,
//     in InstanceStateMask instance_states);

```

```
//
// ReturnCode_t
// take_instance(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in InstanceHandle_t a_handle,
//     in SampleStateMask sample_states,
//     in ViewStateMask view_states,
//     in InstanceStateMask instance_states);
//
// ReturnCode_t
// read_next_instance(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in InstanceHandle_t a_handle,
//     in SampleStateMask sample_states,
//     in ViewStateMask view_states,
//     in InstanceStateMask instance_states);
//
// ReturnCode_t
// take_next_instance(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in InstanceHandle_t a_handle,
//     in SampleStateMask sample_states,
//     in ViewStateMask view_states,
//     in InstanceStateMask instance_states);
//
// ReturnCode_t
// read_next_instance_w_condition(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in InstanceHandle_t a_handle,
//     in ReadCondition a_condition);
//
// ReturnCode_t
// take_next_instance_w_condition(
//     inout DataSeq data_values,
//     inout SampleInfoSeq info_seq,
//     in Long max_samples,
//     in InstanceHandle_t a_handle,
//     in ReadCondition a_condition);
//
// ReturnCode_t
// return_loan(
//     inout DataSeq data_values,
```

```

//      inout SampleInfoSeq info_seq);
//
//  ReturnCode_t
//  get_key_value(
//      inout Data key_holder,
//      in InstanceHandle_t handle);
//
//  InstanceHandle_t
//  lookup_instance(
//      in Data instance);
ReadCondition
create_readcondition(
    in SampleStateMask sample_states,
    in ViewStateMask view_states,
    in InstanceStateMask instance_states);
QueryCondition
create_querycondition(
    in SampleStateMask sample_states,
    in ViewStateMask view_states,
    in InstanceStateMask instance_states,
    in string query_expression,
    in StringSeq query_parameters);
ReturnCode_t
delete_readcondition(
    in ReadCondition a_condition);
ReturnCode_t
delete_contained_entities();
ReturnCode_t
set_qos(
    in DataReaderQos qos);
ReturnCode_t
get_qos(
    inout DataReaderQos qos);
ReturnCode_t
set_listener(
    in DataReaderListener a_listener,
    in StatusMask mask);
DataReaderListener
get_listener();
TopicDescription
get_topicdescription();
Subscriber
get_subscriber();
ReturnCode_t
get_sample_rejected_status(
    inout SampleRejectedStatus status);
ReturnCode_t
get_liveliness_changed_status(
    inout LivelinessChangedStatus status);
ReturnCode_t

```

```

get_requested_deadline_missed_status(
    inout RequestedDeadlineMissedStatus status);
ReturnCode_t
get_requested_incompatible_qos_status(
    inout RequestedIncompatibleQosStatus status);
ReturnCode_t
get_subscription_matched_status(
    inout SubscriptionMatchedStatus status);
ReturnCode_t
get_sample_lost_status(
    inout SampleLostStatus status);
ReturnCode_t
wait_for_historical_data(
    in Duration_t max_wait);
ReturnCode_t
get_matched_publications(
    inout InstanceHandleSeq publication_handles);
ReturnCode_t
get_matched_publication_data(
    inout PublicationBuiltinTopicData publication_data,
    in InstanceHandle_t publication_handle);
};
struct SampleInfo {
SampleStateKind sample_state;
ViewStateKind view_state;
InstanceStateKind instance_state;
Time_t source_timestamp;
InstanceHandle_t instance_handle;
BuiltinTopicKey_t publication_handle;
long disposed_generation_count;
long no_writers_generation_count;
long sample_rank;
long generation_rank;
long absolute_generation_rank;
boolean valid_data;
};
typedef sequence<SampleInfo> SampleInfoSeq;
};
Foo.idl
// Implied IDL for type "Foo"
// Example user defined structure
struct Foo {
long dummy;
};
typedef sequence<Foo> FooSeq;
#include "dds_dcps.idl"
interface FooTypeSupport : DDS::TypeSupport {
DDS::ReturnCode_t
register_type(
    in DDS::DomainParticipant participant,

```

```

        in string type_name);
string
get_type_name();
};
interface FooDataWriter : DDS::DataWriter {
DDS::InstanceHandle_t
register_instance(
    in Foo instance_data);
DDS::InstanceHandle_t
register_instance_w_timestamp(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle,
    in DDS::Time_t source_timestamp);
DDS::ReturnCode_t
unregister_instance(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle);
DDS::ReturnCode_t
unregister_instance_w_timestamp(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle,
    in DDS::Time_t source_timestamp);
DDS::ReturnCode_t
write(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle);
DDS::ReturnCode_t
write_w_timestamp(
    in Foo instance_data,
    in DDS::InstanceHandle_t handle,
    in DDS::Time_t source_timestamp);
DDS::ReturnCode_t
dispose(
    in Foo instance_data,
    in DDS::InstanceHandle_t instance_handle);
DDS::ReturnCode_t
dispose_w_timestamp(
    in Foo instance_data,
    in DDS::InstanceHandle_t instance_handle,
    in DDS::Time_t source_timestamp);
DDS::ReturnCode_t
get_key_value(
    inout Foo key_holder,
    in DDS::InstanceHandle_t handle);
DDS::InstanceHandle_t
lookup_instance(
    in Foo instance_data);
};
interface FooDataReader : DDS::DataReader {
DDS::ReturnCode_t

```

```

read(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);
DDS::ReturnCode_t
take(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);
DDS::ReturnCode_t
read_w_condition(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::ReadCondition a_condition);
DDS::ReturnCode_t
take_w_condition(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::ReadCondition a_condition);
DDS::ReturnCode_t
read_next_sample(
    inout Foo data_values,
    inout DDS::SampleInfo sample_info);
DDS::ReturnCode_t
take_next_sample(
    inout Foo data_values,
    inout DDS::SampleInfo sample_info);
DDS::ReturnCode_t
read_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);
DDS::ReturnCode_t
take_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::InstanceHandle_t a_handle,

```

```

        in DDS::SampleStateMask sample_states,
        in DDS::ViewStateMask view_states,
        in DDS::InstanceStateMask instance_states);
DDS::ReturnCode_t
read_next_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);
DDS::ReturnCode_t
take_next_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);
DDS::ReturnCode_t
read_next_instance_w_condition(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::ReadCondition a_condition);
DDS::ReturnCode_t
take_next_instance_w_condition(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq,
    in Long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::ReadCondition a_condition);
DDS::ReturnCode_t
return_loan(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq info_seq);
DDS::ReturnCode_t
get_key_value(
    inout Foo key_holder,
    in DDS::InstanceHandle_t handle);
DDS::InstanceHandle_t
lookup_instance(
    in Foo instance);
};

```


D *SampleStates, ViewStates and InstanceStates*

Data is made available to the application by the following operations on `DataReader` objects: `read` and `take` operations. The general semantics of the `read` operations is that the application only gets access to the matching data; the data remain available in the Data Distribution Services and can be read again. The semantics of the `take` operations is that the data is not available in the Data Distribution Service; that data will no longer be accessible to the `DataReader`. Consequently, it is possible for a `DataReader` to access the same sample multiple times but only if all previous accesses were `read` operations.

Each of these operations returns an ordered collection of `Data` values and associated `SampleInfo` objects. Each data value represents an atom of data information (i.e., a value for one instance). This collection may contain samples related to the same or different instances (identified by the `key`). Multiple samples can refer to the same instance if the settings of the `HistoryQosPolicy` allow for it.

SampleInfo Class

`SampleInfo` is the information that accompanies each sample that is ‘read’ or ‘taken’. It contains, among others, the following information:

- The `sample_state` (`READ_SAMPLE_STATE` or `NOT_READ_SAMPZWLE_STATE`)
- The `view_state` (`NEW_VIEW_STATE` or `NOT_NEW_VIEW_STATE`)
- The `instance_state` (`ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE` or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`).

sample_state

For each sample, the Data Distribution Service internally maintains a `sample_state` specific to each `DataReader`. The `sample_state` can either be `READ_SAMPLE_STATE` or `NOT_READ_SAMPLE_STATE`.

`READ_SAMPLE_STATE` indicates that the `DataReader` has already accessed that sample by means of `read`. Had the sample been accessed by `take` it would no longer be available to the `DataReader`;

- NOT_READ_SAMPLE_STATE indicates that the DataReader has not accessed that sample before.

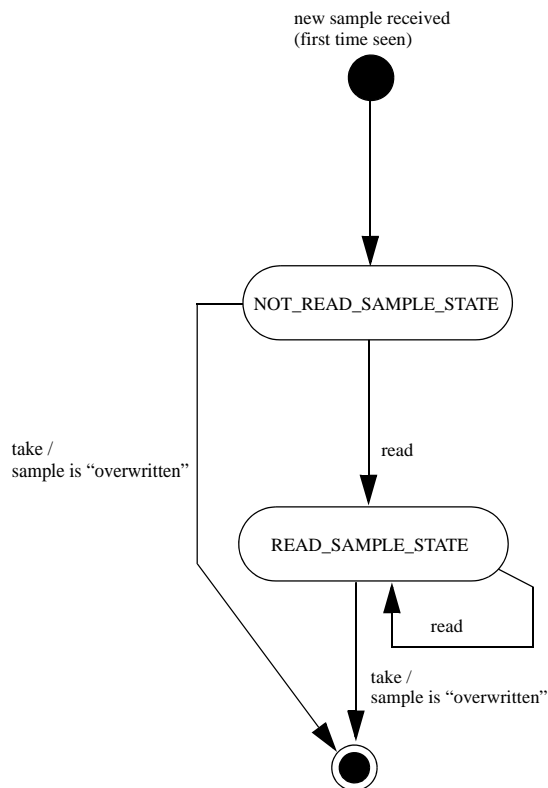


Figure 20: Single Sample sample_state State Chart

State Per Sample

The sample_state available in the SampleInfo reflect the sample_state of each sample. The sample_state can be different for all samples in the returned collection that refer to the same instance.

instance_state

For each instance the Data Distribution Service internally maintains an instance_state. The instance_state can be:

- ALIVE_INSTANCE_STATE, which indicates that
 - samples have been received for the instance

- there are live `DataWriter` objects writing the instance
- the instance has not been explicitly disposed of (or else samples have been received after it was disposed of)
- `NOT_ALIVE_DISPOSED_INSTANCE_STATE` indicates the instance was disposed of by a `DataWriter`, either explicitly by means of the `dispose` operation or implicitly in case the `autodispose_unregistered_instances` field of the `WriterDataLifecycleQosPolicy` equals `TRUE` when the instance gets unregistered (see Section 3.1.3.23, *WriterDataLifecycleQosPolicy*) and no new samples for that instance have been written afterwards.
- `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE` indicates the instance has been declared as not-alive by the `DataReader` because it detected that there are no live `DataWriter` objects writing that instance.

OwnershipQosPolicy

The precise events that cause the `instance_state` to change depends on the setting of the `OwnershipQosPolicy`:

- If `OwnershipQosPolicy` is set to `EXCLUSIVE_OWNERSHIP_QOS`, then the `instance_state` becomes `NOT_ALIVE_DISPOSED_INSTANCE_STATE` only if the `DataWriter` that “owns” the instance explicitly disposes of it. The `instance_state` becomes `ALIVE_INSTANCE_STATE` again only if the `DataWriter` that owns the instance writes it;
- If `OwnershipQosPolicy` is set to `SHARED_OWNERSHIP_QOS`, then the `instance_state` becomes `NOT_ALIVE_DISPOSED_INSTANCE_STATE` if any `DataWriter` explicitly disposes of the instance. The `instance_state` becomes `ALIVE_INSTANCE_STATE` as soon as any `DataWriter` writes the instance again.

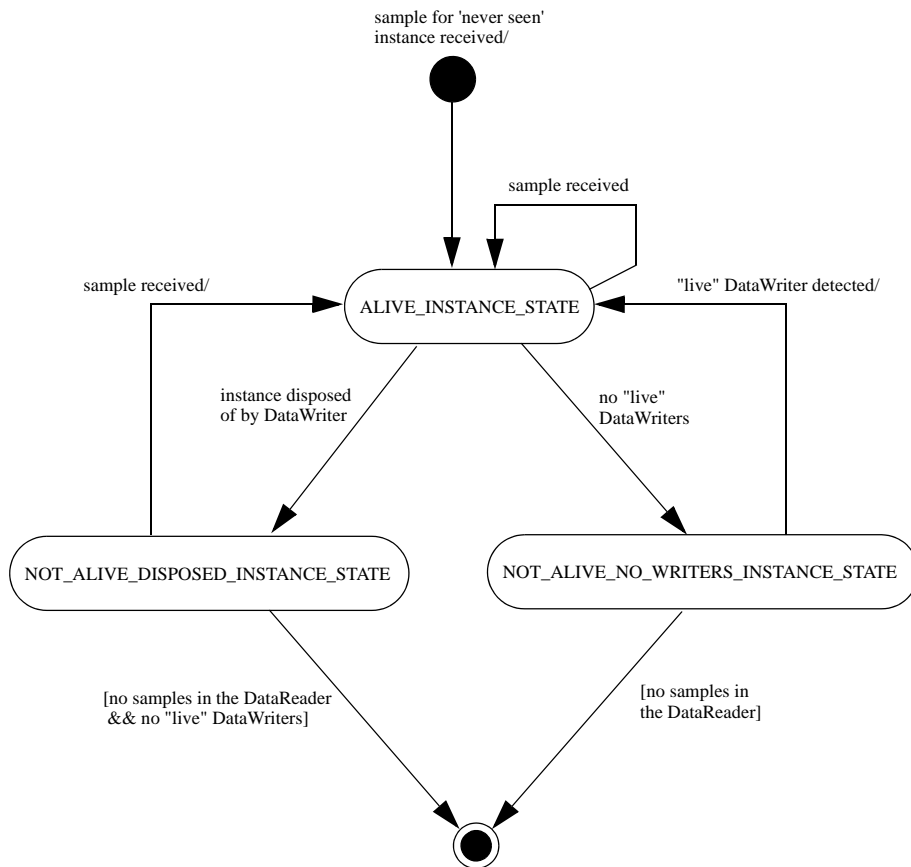


Figure 21: State Chart of the instance_state for a Single Instance

Snapshot

The `instance_state` available in the `SampleInfo` is a snapshot of the `instance_state` of the instance at the time the collection was obtained (i.e. at the time `read` or `take` was called). The `instance_state` is therefore the same for all samples in the returned collection that refer to the same instance.

view_state

For each instance (identified by the `key`), the Data Distribution Service internally maintains a `view_state` relative to each `DataReader`. The `view_state` can either be `NEW_VIEW_STATE` or `NOT_NEW_VIEW_STATE`.

`NEW_VIEW_STATE` indicates that either this is the first time that the `DataReader` has ever accessed samples of that instance, or else that the `DataReader` has accessed previous samples of the instance, but the instance has since been reborn (i.e. becomes not-alive and then alive again);

- `NOT_NEW_VIEW_STATE` indicates that the `DataReader` has already accessed samples of the same instance and that the instance has not been reborn since.

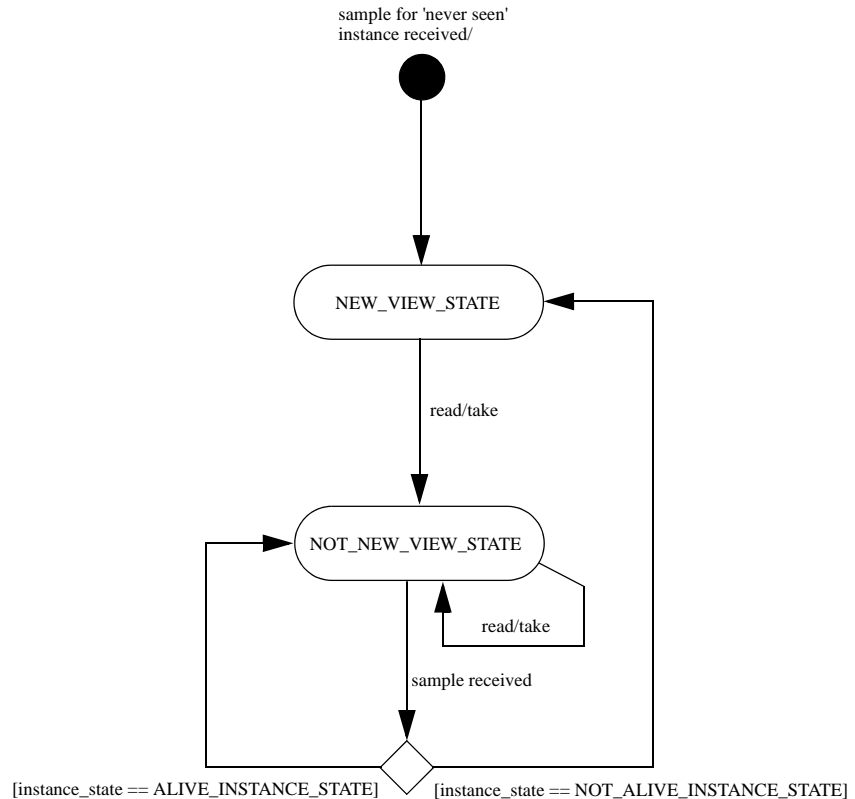


Figure 22: Single Instance `view_state` State Chart

Snapshot

The `view_state` available in the `SampleInfo` is a snapshot of `view_state` of the instance relative to the `DataReader` used to access the samples at the time the collection was obtained (i.e. at the time `read` or `take` was called). The `view_state` is therefore the same for all samples in the returned collection that refer to the same instance.

State Masks

State Definitions

All states are available as a constant. These convenience constants can be used to create a bit mask (e.g. to be used as operation parameters) by performing an AND or OR operation. They can also be used for testing whether a state is set.

The sample state definitions indicates whether or not the matching data sample has already been read:

- `READ_SAMPLE_STATE`: sample has already been read;
- `NOT_READ_SAMPLE_STATE`: sample has not been read.

The view state definitions indicates whether the `DataReader` has already seen samples for the most-current generation of the related instance:

- `NEW_VIEW_STATE`: all samples of this instance are new;
- `NOT_NEW_VIEW_STATE`: some or all samples of this instance are not new.

The instance state definitions indicates whether the instance is currently in existence or, if it has been disposed of, the reason why it was disposed of:

- `ALIVE_INSTANCE_STATE`: this instance is currently in existence;
- `NOT_ALIVE_DISPOSED_INSTANCE_STATE`: this instance was disposed of by a `DataWriter`;
- `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`: the instance has been disposed of by the `DataReader` because none of the `DataWriter` objects currently “alive” (according to the `LivelinessQosPolicy`) are writing the instance.

Pre-defined Bit Mask Definitions

For convenience, some pre-defined bit masks are available as a constant definition. These bit mask constants can be used where a state bit mask is required. They can also be used for testing whether certain bits are set.

The sample state bit mask definition selects both sample states

- `ANY_SAMPLE_STATE`: either the sample has already been read or not read

The view state bit mask definition selects both view states

- `ANY_VIEW_STATE`: either the sample has already been seen or not seen

The instance state bit mask definitions selects a combination of instance states

- `NOT_ALIVE_INSTANCE_STATE`: this instance was disposed of by a `DataWriter` or the `DataReader`
- `ANY_INSTANCE_STATE`: this Instance is either in existence or not in existence

Operations Concerning States

The application accesses data by means of the operations `read` or `take` on the `DataReader`. These operations return an ordered collection of `DataSamples` consisting of a `SampleInfo` part and a `Data` part. The way the Data Distribution Service builds this collection (i.e., the data-samples that are parts of the list as well as their order) depends on `QosPolicy` settings set on the `DataReader` and the `Subscriber`, as well as the source timestamp of the samples and the parameters passed to the `read/take` operations, namely:

- the desired sample states (in other words, `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE`, or `ANY_SAMPLE_STATE`)
- the desired view states (in other words, `NEW_VIEW_STATE`, `NOT_NEW_VIEW_STATE`, or `ANY_VIEW_STATE`)
- the desired instance states
`ALIVE_INSTANCE_STATE`,
`NOT_ALIVE_DISPOSED_INSTANCE_STATE`,
`NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`,
`NOT_ALIVE_INSTANCE_STATE`, or `ANY_INSTANCE_STATE`).

The `read` and `take` operations are non-blocking and just deliver what is currently available that matches the specified states.

On output, the collection of `Data` values and the collection of `SampleInfo` structures are of the same length and are in a one-to-one correspondence. Each `SampleInfo` provides information, such as the `source_timestamp`, the `sample_state`, `view_state`, and `instance_state`, etc., about the matching sample.

Some elements in the returned collection may not have valid data. If the `instance_state` in the `SampleInfo` is `NOT_ALIVE_DISPOSED_INSTANCE_STATE` or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`, then the last sample for that instance in the collection, that is, the one whose `SampleInfo` has `sample_rank==0` does not contain valid data. Samples that contain no data do not count towards the limits imposed by the `ResourceLimitsQosPolicy`.

read

The act of reading a sample sets its `sample_state` to `READ_SAMPLE_STATE`. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

take

The act of taking a sample removes it from the `DataReader` so it cannot be ‘read’ or ‘taken’ again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

read_w_condition

In case the `ReadCondition` is a ‘plain’ `ReadCondition` and not the specialized `QueryCondition`, the operation is equivalent to calling `read` and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in the `ReadCondition`. Using this operation the application can avoid repeating the same parameters specified when creating the `ReadCondition`.

take_w_condition

The act of taking a sample removes it from the `DataReader` so it cannot be ‘read’ or ‘taken’ again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

In case the `ReadCondition` is a ‘plain’ `ReadCondition` and not the specialized `QueryCondition`, the operation is equivalent to calling `take` and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in the `ReadCondition`. Using this operation the application can avoid repeating the same parameters specified when creating the `ReadCondition`.

read_next_sample

The `read_next_sample` operation is semantically equivalent to the `read` operation where the input Data sequence has `max_len=1`, the `sample_states=NOT_READ_SAMPLE_STATE`, the `view_states=ANY_VIEW_STATE`, and the `instance_states=ANY_INSTANCE_STATE`.

take_next_sample

The `take_next_sample` operation is semantically equivalent to the `take` operation where the input sequence has `max_len=1`, the `sample_states=NOT_READ_SAMPLE_STATE`, the `view_states=ANY_VIEW_STATE`, and the `instance_states=ANY_INSTANCE_STATE`.

read_instance

The act of reading a sample sets its `sample_state` to `READ_SAMPLE_STATE`. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

take_instance

The act of taking a sample removes it from the `DataReader` so it cannot be ‘read’ or ‘taken’ again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.

Appendix

E Class Inheritance

This appendix gives an overview of the inheritance relations of the DCPS classes.

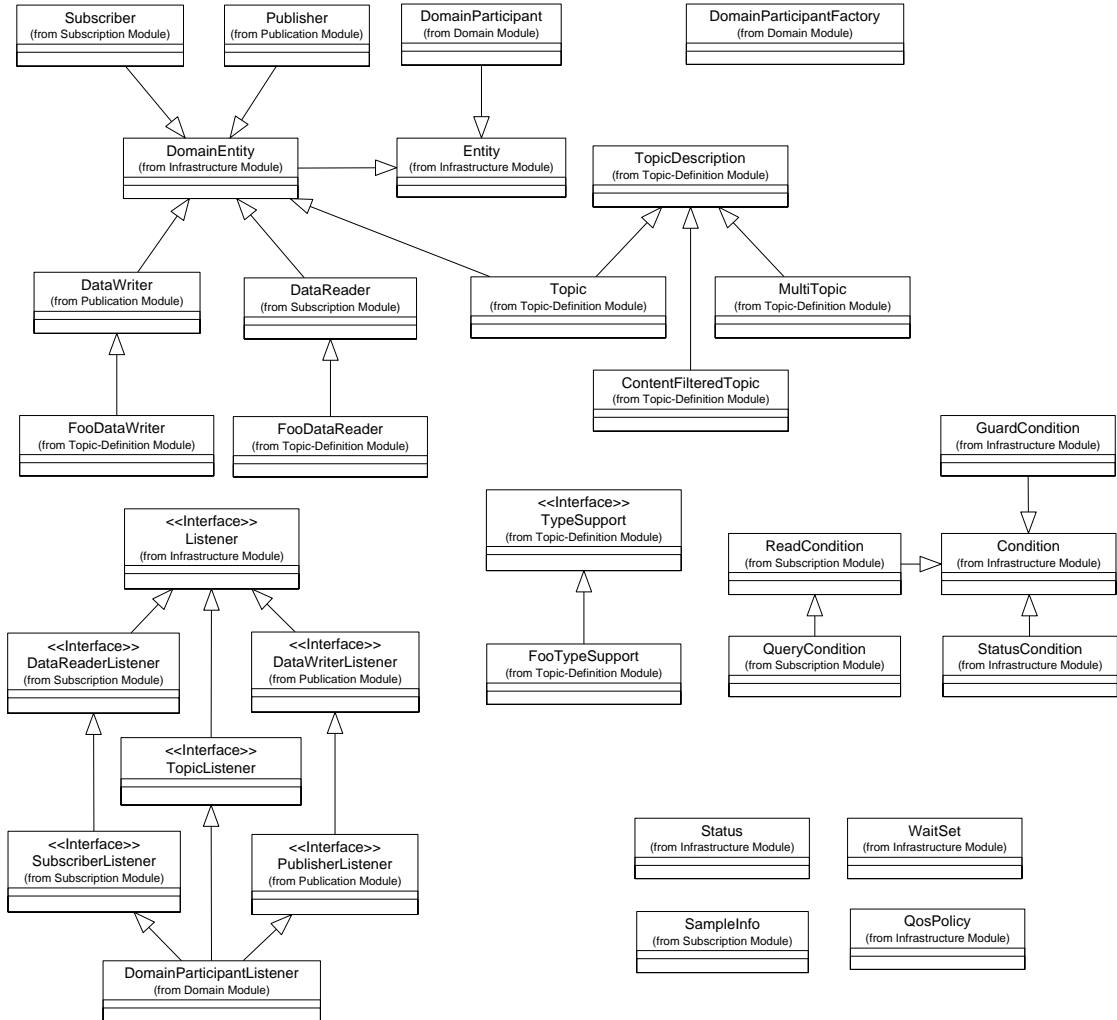


Figure 23 DCPS Inheritance

F

Listeners, Conditions and Waitsets

Listeners and Conditions (Conditions in conjunction with WaitSets) are two mechanisms that allow the application to be made aware of changes in the communication status. Listeners provide an event-based mechanism for the Data Distribution Service to asynchronously alert the application of the occurrence of relevant status changes. Conditions in conjunction with WaitSets provide a state-based mechanism for the Data Distribution Service to synchronously communicate the relevant status changes to the application.

Both mechanisms are based on the communication statuses associated with an Entity object. Not all statuses are applicable to all Entity objects. Which status is applicable to which Entity object is listed in the next table.:

Table 22 Communication States

Entity	Status Name	Description
Topic	INCONSISTENT_TOPIC_STATUS	Another Topic exists with the same name but with different characteristics.
Subscriber	DATA_ON_READERS_STATUS	New information is available.
DataReader	SAMPLE_REJECTED_STATUS	A (received) sample has been rejected.
	LIVELINESS_CHANGED_STATUS	The liveliness of one or more DataWriter objects, that were writing instances read through the DataReader objects has changed. Some DataWriter object have become “alive” or “not alive”.
	REQUESTED_DEADLINE_MISSED_STATUS	The deadline that the DataReader was expecting through its DeadlineQoSPolicy was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS_STATUS	A QoSPolicy setting was incompatible with what is offered.
	DATA_AVAILABLE_STATUS	New information is available.
	SAMPLE_LOST_STATUS	A sample has been lost (never received).
	SUBSCRIPTION_MATCH_STATUS	The DataReader has found a DataWriter that matches the Topic and has compatible QoS.

Table 22 Communication States (Continued)

Entity	Status Name	Description
DataWriter	LIVELINESS_LOST_STATUS	The liveliness that the DataWriter has committed through its LivelinessQosPolicy was not respected; thus DataReader objects will consider the DataWriter as no longer “active”.
	OFFERED_DEADLINE_MISSED_STATUS	The deadline that the DataWriter has committed through its DeadlineQosPolicy was not respected for a specific instance.
	OFFERED_INCOMPATIBLE_QOS_STATUS	A QosPolicy setting was incompatible with what was requested.
	PUBLICATION_MATCH_STATUS	The DataWriter has found DataReader that matches the Topic and has compatible QoS.

The statuses may be classified in:

- *read communication statuses*: i.e., those that are related to arrival of data, namely `DATA_ON_READERS` and `DATA_AVAILABLE`;
- *plain communication statuses*: i.e., all the others.

For each plain communication status, there is a corresponding status struct. The information from this struct can be retrieved with the operations `get_<status_name>_status`. For example, to get the `INCONSISTENT_TOPIC` status (which information is stored in the `InconsistentTopicStatus` struct), the application must call the operation `get_inconsistent_topic_status`. A plain communication status can only be read from the Entity on which it is applicable. For the read communication statuses there is no struct available to the application.

Communication Status Event

Conceptually associated with each Entity communication status is a logical `StatusChangedFlag`. This flag indicates whether that particular communication status has changed since the last time the status was ‘read’ by the application (there is no actual read-operation to read the `StatusChangedFlag`). The `StatusChangedFlag` is only conceptually needed to explain the behaviour of a Listener, therefore, it is not important whether this flag actually exists. A Listener will only be activated when the `StatusChangedFlag` changes from `FALSE` to `TRUE` (provided the Listener is attached and enabled for this particular status). The conditions which cause the `StatusChangedFlag` to change is slightly different for the plain communication status and the read communication status.

For the plain communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` whenever the plain communication status changes and it is reset to `FALSE` each time the application accesses the plain communication status via the proper `get_<status_name>_status` operation on the `Entity`.

The communication status is also reset to `FALSE` whenever the associated `Listener` operation is called as the `Listener` implicitly accesses the status which is passed as a parameter to the operation. The fact that the status is reset prior to calling the listener means that if the application calls the `get_<status_name>_status` from inside the listener it will see the status already reset.

An exception to this rule is when the associated `Listener` is the 'nil' listener, i.e. a listener with value `NULL`. Such a listener is treated as a `NOOP`¹ for all statuses activated in its bit mask and the act of calling this 'nil' listener does not reset the corresponding communication statuses.

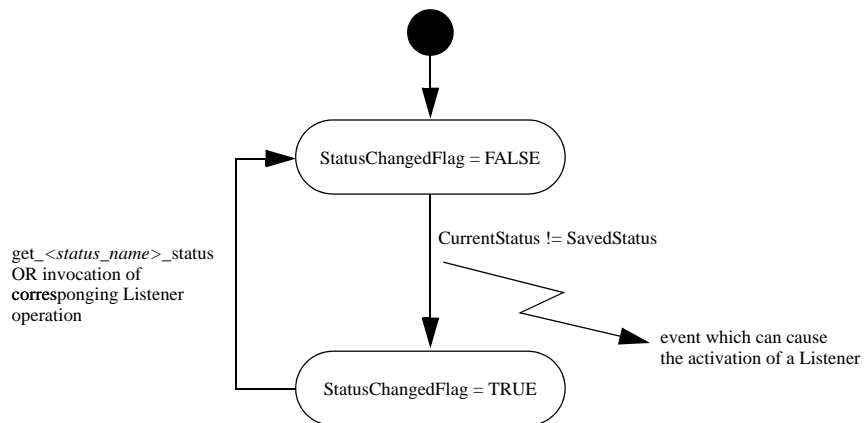


Figure 24: Plain Communication Status State Chart

For example, the value of the `StatusChangedFlag` associated with the `RequestedDeadlineMissedStatus` will become `TRUE` each time a new deadline passes (which increases the `total_count` field within `RequestedDeadlineMissedStatus`). The value changes to `FALSE` when the application accesses the status via the corresponding `get_requested_deadline_missed_status` operation on the proper `Entity`, or when the `on_requested_deadline_missed` operation on the `Listener` attached to this `Entity` or one its containing entities is invoked.

1. Short for **No-Operation**, an instruction that does nothing.

For the read communication status, the `StatusChangedFlag` flag is initially set to `FALSE`. It becomes `TRUE` when data arrives, or when the `InstanceState` of a contained instance changes. This can be caused by either:

- The arrival of the notification that an instance has been disposed by:
 - the `DataWriter` that owns it if its `OwnershipQosPolicyKind = EXCLUSIVE_OWNERSHIP_QOS`
 - or by any `DataWriter` if its `OwnershipQosPolicyKind = SHARED_OWNERSHIP_QOS`.
- The loss of liveness of the `DataWriter` of an instance for which there is no other `DataWriter`.
- The arrival of the notification that an instance has been unregistered by the only `DataWriter` that is known to be writing the instance.

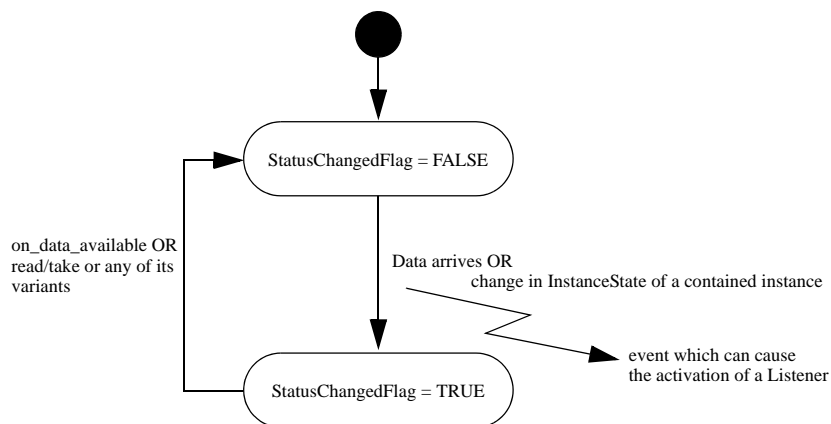


Figure 25: Read Communication Status DataReader Statecraft

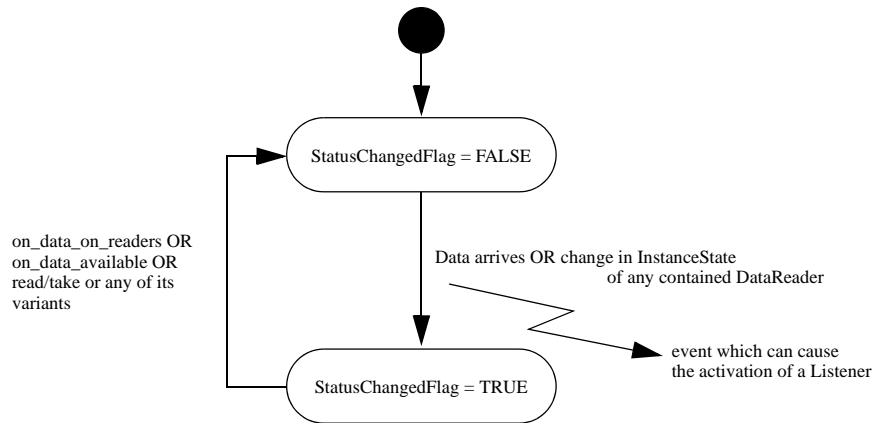


Figure 26: Subscriber Statecraft for a Read Communication Status

- The status flag of the `DATA_ON_READERS_STATUS` becomes `FALSE` when any of the following events occurs:
 - The corresponding listener operation (`on_data_on_readers`) is called on the corresponding Subscriber.
 - The `on_data_available` listener operation is called on any `DataReader` belonging to the Subscriber.
 - The `read` or `take` operation (or any of its variants) is called on any `DataReader` belonging to the Subscriber.

Listeners

The `Listeners` provide for an event-based mechanism to asynchronously inform the application of a status change event. `Listeners` are applicable for both the read communication statuses and the plain communication statuses. When one of these status change events occur, the associated `Listener` is activated, provided some pre-conditions are satisfied. When the `Listener` is activated, it will call the corresponding `on_<status_name>` operation of that `Listener`. Each `on_<status_name>` operation available in the `Listener` of an `Entity` is also available in the `Listener` of the factory of the `Entity`.

For both the read communication statuses and the plain communication statuses a `Listener` is only activated when a `Listener` is attached to this particular `Entity` and enabled for this particular status. Statuses are enabled according to the `StatusKindMask` parameter that was passed at creation time of the `Entity`, or that was passed to the `set_listener` operation.

When an event occurs for a particular Entity and for a particular status, but the applicable Listener is not activated for this status, the status is propagated up to the factory of this Entity. For this factory, the same propagation rules apply. When even the DomainParticipantListener is not attached or enabled for this status, the application will not be notified about this event. This means, that a status change on a contained Entity only invokes the Listener of its factory if the Listener of the contained Entity itself does not handle the trigger event generated by the status change.

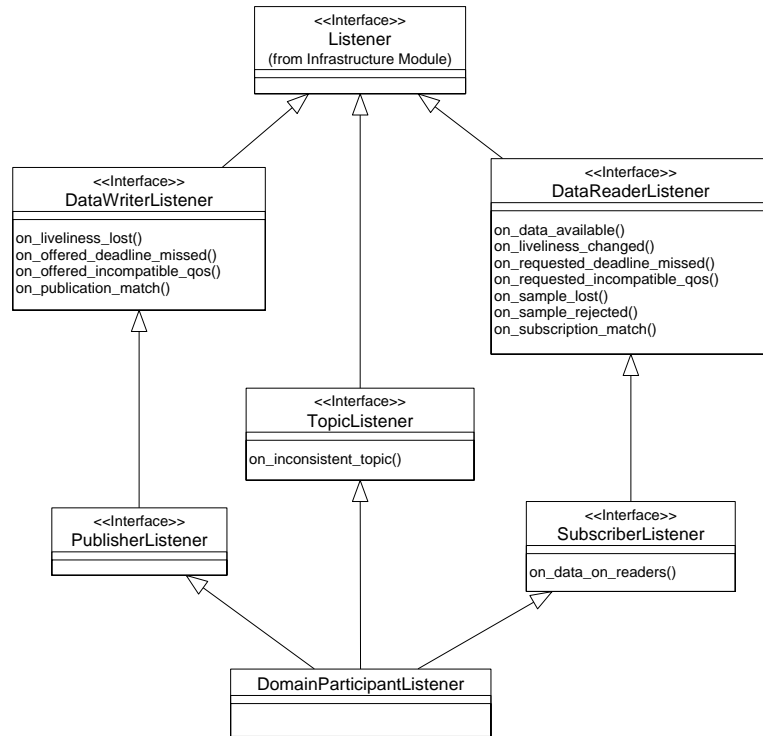


Figure 27: DCPS Listeners

The event propagation is also applicable to the read communication statuses. However, since the event here is the arrival of data, both the DATA_ON_READERS and DATA_AVAILABLE status are TRUE. The Data Distribution Service will first attempt to handle the DATA_ON_READERS status and try to activate the SubscriberListener. When this Listener is not activated for this status the event will propagate to the DomainParticipantListener. Only when the DATA_ON_READERS status can not be handled, the Data Distribution Service will

attempt to handle the `DATA_AVAILABLE` status and try to activate the `DataReaderListener`. In case this `Listener` is not activated for this status the event will follow the propagation rules as described above.

Conditions and Waitsets

The `Conditions` in conjunction with `WaitSets` provide for a state-based mechanism to synchronously inform the application of status changes. A `Condition` can be either a `ReadCondition`, `QueryCondition`, `StatusCondition` or `GuardCondition`. To create a `Condition` one of the following operations can be used:

- `ReadCondition` created by `create_readcondition`
- `QueryCondition` created by `create_querycondition`
- `StatusCondition` retrieved by `get_statuscondition` on an `Entity`
- `GuardCondition` created by the C++ operation `new`

Note that the `QueryCondition` is a specialized `ReadCondition`. The `GuardCondition` is a different kind of `Condition` since it is not controlled by a status but directly by the application (when a `GuardCondition` is initially created, the `trigger_value` is `FALSE`). The `StatusCondition` is present by default with each `Entity`, therefore, it does not have to be created.

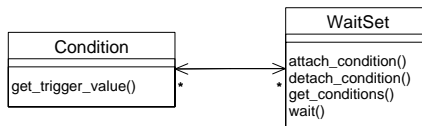


Figure 28: DCPS WaitSets

A `WaitSet` may have one or several `Conditions` attached to it. An application thread may block execution (blocking may be limited by a timeout) by waiting on a `WaitSet` until the `trigger_value` of one or more of the `Conditions` become `TRUE`. When a `Condition`, whose `trigger_value` evaluates to `TRUE`, is attached to a `WaitSet` that is currently being waited on (using the `wait` operation), the `WaitSet` will unblock immediately.

This (state-based) mechanism is generally used as follows:

- The application creates a `WaitSet`.
- The application indicates which relevant information it wants to be notified of, by creating or retrieving `Condition` objects (`StatusCondition`, `ReadCondition`, `QueryCondition` or `GuardCondition`) and attach them to a `WaitSet`.

- It then waits on that WaitSet (using WaitSet::wait) until the trigger_value of one or several Condition objects (in the WaitSet) become TRUE.
- When the thread is unblocked, the application uses the result of the wait (i.e., the list of Condition objects with trigger_value==TRUE) to actually get the information:
 - if the condition is a StatusCondition and the status changes refer to a plain communication status, by calling get_status_changes and then get_<communication_status> on the relevant Entity
 - if the condition is a StatusCondition and the status changes refer to the read communication status:
 - DATA_ON_READERS, by calling get_status_changes and then get_datareaders on the relevant Subscriber and then read/take on the returned DataReader objects
 - DATA_AVAILABLE, by calling get_status_changes and then read/take on the relevant DataReader
 - if it is a ReadCondition or a QueryCondition, by calling directly read_w_condition/take_w_condition on the DataReader with the Condition as a parameter

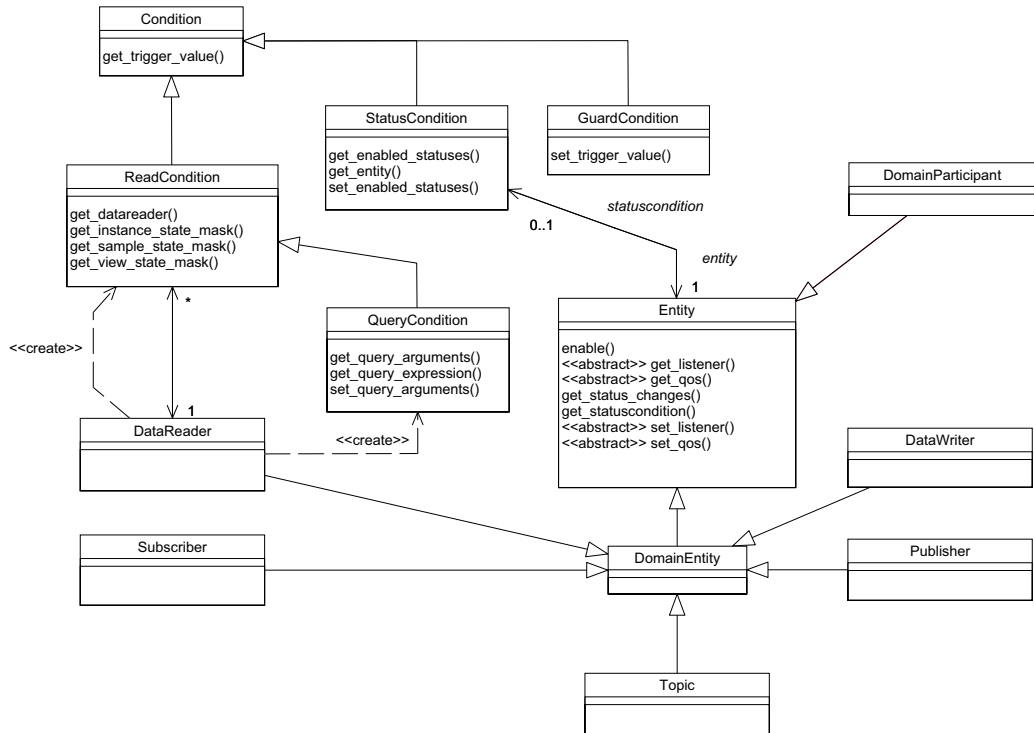


Figure 29 DCPS Conditions

No extra information is passed from the Data Distribution Service to the application when a `wait` returns only the list of triggered `Condition` objects. Therefore, it is the application responsibility to investigate which `Condition` objects have triggered the `WaitSet`.

Blocking Behaviour

The result of a `wait` operation depends on the state of the `WaitSet`, which in turn depends on whether at least one attached `Condition` has a `trigger_value` of `TRUE`. If the `wait` operation is called on `WaitSet` with state `BLOCKED` it will block the calling thread. If `wait` is called on a `WaitSet` with state `UNBLOCKED` it will return immediately. In addition, when the `WaitSet` transitions from state `BLOCKED` to state `UNBLOCKED` it wakes up the thread (if any) that had called `wait` on it. Note that there can only be one thread waiting on a single `WaitSet`.

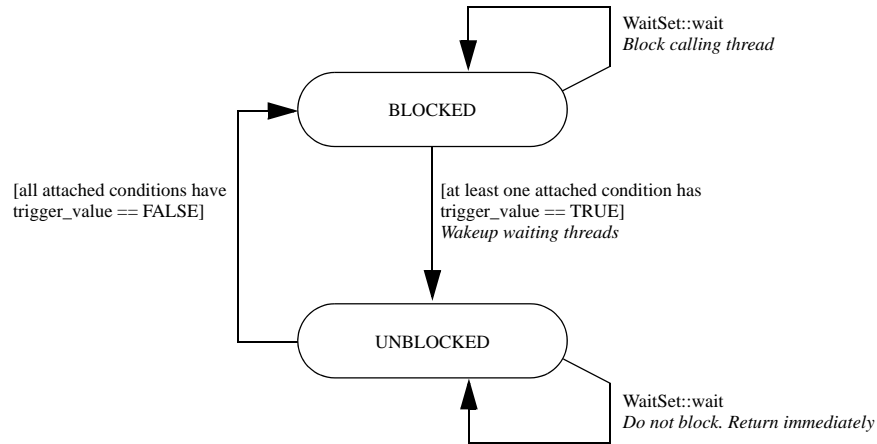


Figure 30: Blocking Behaviour of a Waitset State Chart

StatusCondition Trigger State

The `trigger_value` of a `StatusCondition` is the boolean OR of the `StatusChangedFlag` of all the communication statuses to which it is sensitive. That is, `trigger_value == FALSE` only if all the values of the `StatusChangedFlags` are `FALSE`.

The sensitivity of the `StatusCondition` to a particular communication status is controlled by the bit mask of `enabled_statuses` set on the `Condition` by means of the `set_enabled_statuses` operation.

ReadCondition and QueryCondition Trigger State

Similar to the `StatusCondition`, a `ReadCondition` also has a `trigger_value` that determines whether the attached `WaitSet` is **BLOCKED** or **UNBLOCKED**. However, unlike the `StatusCondition`, the `trigger_value` of the `ReadCondition` is tied to the presence of at least one sample managed by the Data Distribution Service with `SampleState`, `ViewState`, and `InstanceState` matching those of the `ReadCondition`. Additionally, for the `QueryCondition`, the data associated with the sample, must be such that the `query_expression` evaluates to `TRUE`.

The fact that the `trigger_value` of a `ReadCondition` is dependent on the presence of samples on the associated `DataReader` implies that a single take operation can potentially change the `trigger_value` of several `ReadCondition` or `QueryCondition` objects. For example, if all samples are taken, any `ReadCondition` or `QueryCondition` objects associated with the `DataReader` that had their `trigger_value == TRUE` before will see the `trigger_value` change to `FALSE`. Note that this does not guarantee that `WaitSet` objects, that had

those `Condition` objects separately attached to, will not be woken up. Once we have `trigger_value==TRUE` on a `Condition` it may wake up the `WaitSet` it was attached to, the condition transitions to `trigger_value==FALSE` does not 'un-wake up' the `WaitSet` as 'un-wakening' is not possible. The consequence is that an application blocked on a `WaitSet` may return from the wait with a list of `Condition` objects some of which are no longer "active". This is unavoidable if multiple threads are concurrently waiting on separate `WaitSet` objects and taking data associated with the same `DataReader` Entity. In other words, a wait may return with a list of `Condition` objects which all have a `trigger_value==FALSE`. This only means, that at some point one or more of the `Condition` objects have had a `trigger_value==TRUE` but no longer do.

GuardCondition Trigger State

The `trigger_value` of a `GuardCondition` is completely controlled by the application via the operation `set_trigger_value`. This `Condition` can be used to implement an application defined wake-up of the blocked thread.

G Topic Definitions

The Data Distribution Service distributes its data in structured data types, called topics. The first step when using the Data Distribution Service consists of defining these topics. Since the Data Distribution Service supports using several programming languages, OMG IDL is used for this purpose. This appendix describes how to define the topics.

Topic Definition Example

All data distributed using the Data Distribution Service has to be defined as a topic. A topic is a structured data type, like a C++-struct with several members. Whenever the application needs to read or write data, it will be reading or writing topics. The definition of each topic it will be using has to be written in (a subset of) OMG IDL. For example:

```
module SPACE {
    struct Foo {
        long      userID; // owner of message
        long long index;  // message index per owner
        string     content; // message body
    };
    #pragma keylist Foo
};
```

This is the definition of a topic called `Foo`, used for sending and receiving messages (as an example). Even though the topic is defined using IDL, the Data Distribution Service will be using an equivalent C++-struct which is accessed by the application using the type specific operations. Generation of the typed classes is achieved by invoking the Data Distribution Service IDL pre-processor: **`idlpp -l c++ -s <idl_filename>.idl`**, a tool which translates the IDL topic definition into an equivalent C++-definition. The `-l c++` option indicates that the C++ code has to be generated (in accordance with the *OMG C++ Language Mapping Specification*). The `-s` option indicates that this C-code should be *StandAlone* C++ code, in other words, it must not have any dependency on external ORB libraries. (It is also possible to use libraries from an existing ORB, so that your DDS application can also manage information coming from an external ORB. In that case you should use the CORBA-cohabitation mode, replacing the `-s` flag with a `-C` flag.). In this example, the pre-processor will generate the classes `FooTypeSupport`, `FooDataWriter` and `FooDataReader` which contain the type specific operations.

Complex Topics

The `Foo` topic is relatively simple, but the Data Distribution Service is capable of distributing more complex topics as well. In fact, any definition following the OpenSplice IDL subset is allowed. It is important to know that the pre-processor accepts all IDL constructs but only the subset is being processed.

Apart from the trivial data types, the Data Distribution Service is capable of handling fixed-length arrays, bounded and unbounded sequences, union types and enumerations. Types can be nested, e.g. a struct can contain a struct field or an array of structs, or a sequence of strings or an array of sequences containing structs.

IDL Pre-processor

This section contains the specification of the subset of OMG IDL that can be used to define the topics.

IDL-to-Host Language Mapping

The Data Distribution Service IDL pre-processor translates the IDL-definition of the topics into language specific code. This translation is executed according to the OMG IDL mappings. Since the Data Distribution Service uses data-structures only, not all IDL-features are implemented by the pre-processor. Usually, the IDL definition consists of a module defining several structs and typedefs.

Data Distribution Service IDL Keywords

The identifiers listed in this appendix are reserved for use as keywords in IDL and may not be used otherwise, unless escaped with a leading underscore.

<code>abstract</code>	<code>exception</code>	<code>inout</code>	<code>provides</code>	<code>truncatable</code>
<code>any</code>	<code>emits</code>	<code>interface</code>	<code>public</code>	<code>typedef</code>
<code>attribute</code>	<code>enum</code>	<code>local</code>	<code>publishes</code>	<code>typeid</code>
<code>boolean</code>	<code>eventtype</code>	<code>long</code>	<code>raises</code>	<code>typeprefix</code>
<code>case</code>	<code>factory</code>	<code>module</code>	<code>readonly</code>	<code>unsigned</code>
<code>char</code>	<code>FALSE</code>	<code>multiple</code>	<code>setraises</code>	<code>union</code>
<code>component</code>	<code>finder</code>	<code>native</code>	<code>sequence</code>	<code>uses</code>
<code>const</code>	<code>fixed</code>	<code>Object</code>	<code>short</code>	<code>ValueBase</code>
<code>consumes</code>	<code>float</code>	<code>octet</code>	<code>string</code>	<code>valuetype</code>
<code>context</code>	<code>getraises</code>	<code>oneway</code>	<code>struct</code>	<code>void</code>
<code>custom</code>	<code>home</code>	<code>out</code>	<code>supports</code>	<code>wchar</code>
<code>default</code>	<code>import</code>	<code>primarykey</code>	<code>switch</code>	<code>wstring</code>
<code>double</code>	<code>in</code>	<code>private</code>	<code>TRUE</code>	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords are illegal. For example, `boolean` is a valid keyword; `Boolean` and `BOOLEAN` are illegal identifiers.

Data Distribution Service IDL Pragma Keylist

To define a topic, the content must either be a struct or a union. The pre-processor will only generate the type specific classes when topic definition is accompanied by a `<pragmakeylist>`. When the `<pragmakeylist>` has no `<field_id>`, the topic is available but no key is set. To define the keylist the definition, written in BNF-notation, is as follows:

```
<pragmakeylist> ::= "#pragma keylist" <type_id> <field_id>*
<type_id> ::= <struct_type_identifier>
           | <union_type_identifier>
<field_id> ::= <member_declarator>
           | <element_spec_declarator>
```

In case of a struct, `<type_id>` is a `<struct_type_identifier>`. In case of a union, `<type_id>` is a `<union_type_identifier>`. The `<struct_type_identifier>` is the identifier used in the struct declaration. The `<union_type_identifier>` is the identifier used in the union declaration. The `<field_id>` is the identifier of a field in the struct or union identified by `<type_id>`. In case of a struct, `<field_id>` is a `<member_declarator>` which is one of the declarators used in the struct member. In case of a union, `<field_id>` is a `<element_spec_declarator>` which is one of the declarators used in the element specification in a case of the union.

For example, for the `Foo` example the next pragma must be used to have the pre-processor generate the typed classes (`FooTypeSupport`, `FooDataWriter` and `FooDataReader`).

```
#pragma keylist Foo userID index
```

Note that in this example the `userID` and the `index` are used as a key.

Data Distribution Service IDL subset in BNF-notation

Only a subset is used by the pre-processor. A description of the Data Distribution Service IDL subset, written in BNF-notation, is as shown below.

```
<definition> ::= <type_dcl> ";"
              | <const_dcl> ";"
              | <module> ";"
<module> ::= "module" <identifier> "{" <definition>+ "}"
<scoped_name> ::= <identifier>
                | "::" <identifier>
                | <scoped_name> "::" <identifier>
```

```

<const_dcl> ::= "const" <const_type>
               <identifier> "=" <const_exp>
<const_type> ::= <integer_type>
               | <char_type>
               | <boolean_type>
               | <floating_pt_type>
               | <string_type>
               | <scoped_name>
               | <octet_type>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr>
             | <or_expr> "|" <xor_expr>
<xor_expr> ::= <and_expr>
             | <xor_expr> "^" <and_expr>
<and_expr> ::= <shift_expr>
             | <and_expr> "&" <shift_expr>
<shift_expr> ::= <add_expr>
               | <shift_expr> ">>" <add_expr>
               | <shift_expr> "<<" <add_expr>
<add_expr> ::= <mult_expr>
               | <add_expr> "+" <mult_expr>
               | <add_expr> "-" <mult_expr>
<mult_expr> ::= <unary_expr>
               | <mult_expr> "*" <unary_expr>
               | <mult_expr> "/" <unary_expr>
               | <mult_expr> "%" <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr>
               | <primary_expr>
<unary_operator> ::= "-"
                 | "+"
                 | "~"
<primary_expr> ::= <scoped_name>
                 | <literal>
                 | "(" <const_exp> ")"
<literal> ::= <integer_literal>
            | <string_literal>
            | <character_literal>
            | <floating_pt_literal>
            | <boolean_literal>
<boolean_literal> ::= "TRUE"
                   | "FALSE"
<positive_int_const> ::= <const_exp>
<type_dcl> ::= "typedef" <type_declarator>
             | <struct_type>
             | <union_type>
             | <enum_type>
<type_declarator> ::= <type_spec> <declarators>
<type_spec> ::= <simple_type_spec>
              | <constr_type_spec>
<simple_type_spec> ::= <base_type_spec>

```

```

        | <template_type_spec>
        | <scoped_name>
<base_type_spec> ::= <floating_pt_type>
        | <integer_type>
        | <char_type>
        | <boolean_type>
        | <octet_type>
<template_type_spec> ::= <sequence_type>
        | <string_type>
<constr_type_spec> ::= <struct_type>
        | <union_type>
        | <enum_type>
<declarators> ::= <declarator> { "," <declarator> }*
<declarator> ::= <simple_declarator>
        | <complex_declarator>
<simple_declarator> ::= <identifier>
<complex_declarator> ::= <array_declarator>
<floating_pt_type> ::= "float"
        | "double"
<integer_type> ::= <signed_int>
        | <unsigned_int>
<signed_int> ::= <signed_short_int>
        | <signed_long_int>
        | <signed_longlong_int>
<signed_short_int> ::= "short"
<signed_long_int> ::= "long"
<signed_longlong_int> ::= "long" "long"
<unsigned_int> ::= <unsigned_short_int>
        | <unsigned_long_int>
        | <unsigned_longlong_int>
<unsigned_short_int> ::= "unsigned" "short"
<unsigned_long_int> ::= "unsigned" "long"
<unsigned_longlong_int> ::= "unsigned" "long" "long"
<char_type> ::= "char"
<boolean_type> ::= "boolean"
<octet_type> ::= "octet"
<struct_type> ::= "struct" <identifier> "{" <member_list> }"
<member_list> ::= <member>+
<member> ::= <type_spec> <declarators> ";"
<union_type> ::= "union" <identifier> "switch"
        "(" <switch_type_spec> ")"
        "{" <switch_body> }"
<switch_type_spec> ::= <integer_type>
        | <char_type>
        | <boolean_type>
        | <enum_type>
        | <scoped_name>
<switch_body> ::= <case>+
<case> ::= <case_label>+ <element_spec> ";"
<case_label> ::= "case" <const_exp> ":"

```

```

| "default" ":"
<element_spec>::= <type_spec> <declarator>
<enum_type>::= "enum" <identifier>
    "{" <enumerator> { "," <enumerator> }* "}"
<enumerator>::= <identifier>
<sequence_type>::= "sequence" "<" <simple_type_spec> ","
    <positive_int_const> ">"
| "sequence" "<" <simple_type_spec> ">"
<string_type>::= "string" "<" <positive_int_const> ">"
| "string"
<array_declarator>::= <identifier> <fixed_array_size>+
<fixed_array_size>::= "[" <positive_int_const> "]"

```

H DCPS Queries and Filters

A subset of SQL syntax is used in several parts of OpenSplice:

- the `filter_expression` in the `ContentFilteredTopic`
- the `topic_expression` in the `MultiTopic`
- the `query_expression` in the `QueryReadCondition`

Those expressions may use a subset of SQL, extended with the possibility to use program variables in the SQL expression. The allowed SQL expressions are defined with the BNF-grammar below. The following notational conventions are made:

- the NonTerminals are typeset in italics
- the 'Terminals' are quoted and typeset in a fixed width font
- the TOKENS are typeset in small caps
- the notation *(element // '')* represents a non-empty comma-separated list of *elements*

SQL Grammar in BNF

```
Expression::= FilterExpression
               | TopicExpression
               | QueryExpression

FilterExpression::= Condition

TopicExpression::= SelectFrom {Where } ';'

QueryExpression::= {Condition}

SelectFrom::= 'SELECT' Aggregation 'FROM' Selection

Aggregation::= '*'
               | (SubjectFieldSpec // ',')

SubjectFieldSpec::= FIELDNAME
                   | FIELDNAME 'AS' FIELDNAME
                   | FIELDNAME FIELDNAME

Selection::= TOPICNAME
             | TOPICNAME NaturalJoin JoinItem

JoinItem::= TOPICNAME
```

```

| TOPICNAME NaturalJoin JoinItem
| '(' TOPICNAME NaturalJoin JoinItem ')'

NaturalJoin ::= 'INNER NATURAL JOIN'
| 'NATURAL JOIN'
| 'NATURAL INNER JOIN'

Where ::= 'WHERE' Condition

Condition ::= Predicate
| Condition 'AND' Condition
| Condition 'OR' Condition
| 'NOT' Condition
| '(' Condition ')'

Predicate ::= ComparisonPredicate
| BetweenPredicate

ComparisonPredicate ::= FIELDNAME RelOp Parameter
| Parameter RelOp FIELDNAME

BetweenPredicate ::= FIELDNAME 'BETWEEN' Range
| FIELDNAME 'NOT BETWEEN' Range

RelOp ::= '=' | '>' | '>=' | '<' | '<=' | '<>' | like

Range ::= Parameter 'AND' Parameter

Parameter ::= INTEGERVALUE
| FLOATVALUE
| STRING
| ENUMERATEDVALUE
| PARAMETER

```



INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. The aliases are all supported because they all are part of the SQL standard.

SQL Token Expression

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

- **FIELDNAME** - A fieldname is a reference to a field in the data-structure. The dot '.' is used to navigate through nested structures. The number of dots that may be used in a fieldname is unlimited. The field-name can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure, which may or may not match the fieldnames that appear on the C mapping of the structure.

- **TOPICNAME** - A topic name is an identifier for a topic, and is defined as any series of characters `'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '-', '_'` but may not start with a digit.
- **INTEGERVALUE** - Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. A hexadecimal number is preceded by `0x` and must be a valid hexadecimal expression.
- **FLOATVALUE** - Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point (`'.'`). A power-of-ten expression may be post-fixed, which has the syntax `en`, where *n* is a number, optionally preceded by a plus or minus sign.
- **STRING** - Any series of characters encapsulated in single quotes, except a new-line character or a right quote. A string starts with a left or right quote, but ends with a right quote.
- **ENUMERATEDVALUE** - An enumerated value is a reference to a value declared within an enumeration. The name of the value must correspond to the names specified in the IDL definition of the enumeration, and must be encapsulated in single quotes. An enum value starts with a left or right quote, but ends with a right quote.
- **PARAMETER** - A parameter is of the form `%n`, where *n* represents a natural number (zero included) smaller than 100. It refers to the *n* + 1th argument in the given context.

Note: when `RelOp` is `'like'`, Unix filename wildcards must be used for strings instead of the normal SQL wildcards. This means any one character is `'?'`, any zero or more characters is `'*'`.

SQL Examples

Assuming Topic “Location” has as an associated type a structure with fields “flight_name, x, y, z”, and Topic “FlightPlan” has as fields “flight_id, source, destination”. The following are examples of using these expressions.

Example of a *topic_expression*:

```
"SELECT flight_name, x, y, z AS height FROM 'Location'
NATURAL JOIN 'FlightPlan' WHERE height < 1000 AND x <23"
```

Example of a *query_expression* or a *filter_expression*:

```
"height < 1000 AND x <23"
```


A close-up, low-angle photograph of a computer keyboard, focusing on the central and right-hand keys. The keys are white with dark lettering. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

BIBLIOGRAPHY

Bibliography

- [1] *OMG Data Distribution Service Revised Final Adopted Specification ptc/04-03-07*, Object Management Group
- [2] *OMG C++ Language Mapping Specification formal/99-07-35*, Object Management Group (OMG)
- [3] *OMG The Common Object Request Broker: Architecture and Specification*, Version 3.0, formal/02-06-01, Object Management Group

A close-up, low-angle photograph of a computer keyboard, focusing on the central and right-hand keys. The keys are white with dark lettering. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

GLOSSARY

Glossary

Acronyms

<i>Acronym</i>	<i>Meaning</i>
CORBA	Common Object Request Broker Architecture
DCPS	Data Centric Publish/Subscribe
DDS	Data Distribution Service
DLRL	Data Local Reconstruction Layer
IDL	Interface Definition Language
OMG	Object Management Group
ORB	Object Request Broker
QoS	Quality of Service
SPLICE	Subscription Paradigm for the Logical Interconnection of Concurrent Engines

A close-up, low-angle photograph of a computer keyboard, focusing on the central and lower-right keys. The keys are white with dark lettering. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

INDEX

Index

A

Affected Entities	401	assert_liveliness (inherited)	249
API Constants and Types	415	Assignment	10
Application Responsibility	467	attach_condition	93
assert_liveliness	111, 229		

B

Basic Usage	401	Bibliography	487
begin_access	289	Blocking Behavior of a Waitset State Chart ..	472
begin_coherent_changes	204	Blocking Behaviour	471

C

C++ Reference Guide Document Structure	3	Class TopicDescription (abstract)	174
Class Condition	98	Class TypeSupport (abstract)	197
Class ContentFilteredTopic	185	Class WaitSet	92
Class DataReader (abstract)	308	Communication States	463
Class DataSample	373	Communication Status Event	464
Class DataWriter (abstract)	226	Complex Topics	476
Class DomainEntity (abstract)	33	Conditions and Waitsets	469
Class DomainParticipant	107	contains_entity	112
Class DomainParticipantFactory	152	copy_from_topic_qos	206, 289
Class Entity (abstract)	26	create_contentfilteredtopic	113
Class FooDataReader	338	create_datareader	291
Class FooDataWriter	246	create_datawriter	207
Class FooTypeSupport	198	create_multitopic	114
Class GuardCondition	99	create_participant	153
Class MultiTopic	190	create_publisher	116
Class Publisher	202	create_querycondition	313
Class QueryCondition	392	create_querycondition (inherited)	343
Class ReadCondition	388	create_readcondition	314
Class StatusCondition	101	create_readcondition (inherited)	343
Class Subscriber	287	create_subscriber	118
Class Topic	177	create_topic	121

D

Data Distribution Service IDL Keywords	476	BNF-notation	477
Data Distribution Service IDL Pragma Keylist ..	477	Data Type “Foo” Typed Classes for Pre-processor	
Data Distribution Service IDL subset in		Generation	20

Data Type “Foo” Typed Classes Pre-processor	
Generation	174
DataReader	466
DataReaderListener Interface	381
DataReaderQos	403
DATAWRITER_QOS_DEFAULT	407
DataWriterListener Interface	281
DataWriterQos	405
DCPS Conditions	98, 471
DCPS Domain Module’s Class Model ...	18, 107
DCPS Infrastructure Module’s Class Model	16, 26
DCPS Inheritance	461
DCPS Listeners	76, 468
DCPS Module Composition	15
DCPS Publication Module’s Class Model	21, 201
DCPS Status Values	78
DCPS Subscription Module’s Class Model	22, 286
DCPS Topic-Definition Module’s Class Model	19, 173
DCPS WaitSets	92, 469
dds_dcps.idl	423
DeadlineQosPolicy	41, 43
delete_contained_entities	124, 210, 294, 315
delete_contained_entities (inherited)	344
delete_contentfilteredtopic	125
delete_datareader	295
delete_datawriter	211
delete_multitopic	126
delete_participant	156
delete_publisher	128
delete_readcondition	317
delete_readcondition (inherited)	344
delete_subscriber	129
delete_topic	130
DestinationOrderQosPolicy	43
detach_condition	94
dispose	250
dispose (abstract)	230
dispose_w_timestamp	253
dispose_w_timestamp (abstract)	231
Document Structure	3
Domain Module	17, 107
DomainParticipantListener interface	167
DomainParticipantQos	409
DurabilityQosPolicy	45
DurabilityServiceQosPolicy	48

E

enable	27
enable (inherited) ..	131, 178, 212, 231, 255, 296, 318, 344
end_access	296
end_coherent_changes	212
EntityFactoryQosPolicy	51

F

find_topic	132
Functionality	15

G

get_builtin_subscriber	133
get_conditions	95
get_current_time	134
get_datareader	389
get_datareader (inherited)	393
get_datareaders	296
get_default_datareader_qos	297
get_default_datawriter_qos	213
get_default_participant_qos	157
get_default_publisher_qos	135
get_default_subscriber_qos	136
get_default_topic_qos	137
get_discovered_participant_data	139
get_discovered_participants	138
get_discovered_topic_data	139
get_discovered_topics	139
get_domain_id	140
get_enabled_statuses	103

get_entity	104	get_qos (inherited)	258, 346
get_expression_parameters	186, 191	get_query_arguments	394
get_filter_expression	187	get_query_expression	395
get_inconsistent_topic_status	178	get_related_topic	188
get_instance	158	get_requested_deadline_missed_status	321
get_instance_handle	29	get_requested_deadline_missed_status (inherited)	346
get_instance_state_mask	390	get_requested_incompatible_qos_status	322
get_instance_state_mask (inherited)	394	get_requested_incompatible_qos_status	(inherited)
get_key_value	255, 344	get_sample_lost_status	323
get_key_value (abstract)	231, 318	get_sample_lost_status (inherited)	346
get_listener	140, 179, 214, 232, 298, 318	get_sample_rejected_status	325
get_listener (abstract)	30	get_sample_rejected_status (inherited)	346
get_listener (inherited)	256, 345	get_sample_state_mask	390
get_liveliness_changed_status	319	get_sample_state_mask (inherited)	396
get_liveliness_changed_status (inherited)	345	get_status_changes	30
get_liveliness_lost_status	232	get_status_changes (inherited) 142, 181, 216, 238,	258,
get_liveliness_lost_status (inherited)	256	get_statuscondition	32
get_matched_publication_data	320	get_statuscondition (inherited) 142, 181, 217, 238,	259,
get_matched_publication_data (inherited)	345	get_subscriber	326
get_matched_publications	320	get_subscriber (inherited)	347
get_matched_publications (inherited)	345	get_subscription_expression	193
get_matched_subscription_data	233	get_subscription_match_status	327
get_matched_subscription_data (inherited)	257	get_subscription_match_status (inherited)	347
get_matched_subscriptions	234	get_topic	238
get_matched_subscriptions (inherited)	257	get_topic (inherited)	259
get_name	175	get_topicdescription	327
get_name (inherited)	180, 188, 192	get_topicdescription (inherited)	348
get_offered_deadline_missed_status	234	get_trigger_value	99
get_offered_deadline_missed_status (inherited)	257	get_trigger_value (inherited)	100, 104, 391, 396
get_offered_incompatible_qos_status	235	get_type_name	176, 199
get_offered_incompatible_qos_status (inherited)	257	get_type_name (abstract)	198
get_participant	175, 215, 299	get_type_name (inherited)	182, 189, 194
get_participant (inherited)	180, 188, 193	get_view_state_mask	391
get_publication_match_status	236	get_view_state_mask (inherited)	396
get_publication_match_status (inherited)	258	GroupDataQosPolicy	51
get_publisher	236	GuardCondition Trigger State	473
get_publisher (inherited)	258		
get_qos	141, 180, 216, 237, 299, 320		
get_qos (abstract)	30		



HistoryQosPolicy 52

I

IDL Pre-processor	476	ignore_topic	143
IDL-to-Host Language Mapping	476	InconsistentTopicStatus	80
ignore_participant	143	Infrastructure Module	16, 26
ignore_publication	143	Inheritance of Abstract Operations	13
ignore_subscription	143	instance_state	452

L

LatencyBudgetQosPolicy	55	LivelinessQosPolicy	57, 59
LifespanQosPolicy	56	lookup_datareader	301
Listener Interface	75	lookup_datawriter	217
Listeners	467	lookup_instance	348
Listeners interfaces	12	lookup_instance (abstract)	328
LivelinessChangedStatus	80	lookup_participant	160
LivelinessLostStatus	82	lookup_topicdescription	144

M

Memory Management	9
-----------------------------	---

N

notify_datareaders	301
------------------------------	-----

O

OfferedDeadlineMissedStatus	83	on_offered_incompatible_qos (inherited, abstract)	
OfferedIncompatibleQosStatus	84	170,	280
on_data_available (abstract)	382	on_publication_match (abstract)	285
on_data_available (inherited, abstract)	169, 378	on_publication_match (inherited, abstract)	171, 281
on_data_on_readers (abstract)	378	on_requested_deadline_missed (abstract)	384
on_data_on_readers (inherited, abstract)	169	on_requested_deadline_missed (inherited, abstract)	380
on_inconsistent_topic (abstract)	196	on_requested_incompatible_qos (abstract)	385
on_inconsistent_topic (inherited, abstract)	169	on_requested_incompatible_qos (inherited, abstract)	171, 380
on_liveliness_changed (abstract)	383	on_sample_lost (abstract)	386
on_liveliness_changed (inherited, abstract)	170, 379	on_sample_lost (inherited, abstract)	171, 380
on_liveliness_lost (abstract)	282	on_sample_rejected (abstract)	387
on_liveliness_lost (inherited, abstract)	170, 280	on_sample_rejected (inherited, abstract)	172, 380
on_offered_deadline_missed (abstract)	283	on_subscription_match (abstract)	388
on_offered_deadline_missed (inherited, abstract)	170,	on_subscription_match (inherited, abstract)	172,
on_offered_incompatible_qos (abstract)	284		

381
 Operations.4
 Operations Concerning States457

OwnershipQosPolicy59, 453
 OwnershipStrengthQosPolicy62

P

PARTICIPANT_QOS_DEFAULT408, 410
 PartitionQosPolicy62
 Plain Communication Status State Chart465
 Pointer Types9
 Pre-defined Bit Mask Definitions.456
 PresentationQosPolicy63

Publication Module20, 201
 Publication Type Specific Classes226
 PublicationMatchStatus86
 PUBLISHER_QOS_DEFAULT411
 PublisherListener Interface279
 PublisherQos410

Q

QosPolicy Basics40
 QosPolicy Default Attributes37

QosPolicy Settings34

R

read349, 457
 read (abstract).328
 read_instance353, 459
 read_instance (abstract)328
 read_next_instance355
 read_next_instance (abstract)329
 read_next_instance_w_condition357
 read_next_instance_w_condition (abstract) . . .329
 read_next_sample359, 458
 read_next_sample (abstract).330
 read_w_condition359, 458
 read_w_condition (abstract).330
 ReadCondition and QueryCondition Trigger State
 472
 ReaderDataLifecycleQosPolicy65
 Reference Count9
 Reference Types9
 register_instance261

register_instance (abstract).239
 register_instance_w_timestamp263
 register_instance_w_timestamp (abstract) . . .240
 register_type200
 register_type (abstract).198
 ReliabilityQosPolicy66
 Requested Offered DestinationOrderQosPolicy 45,
 60
 Requested Offered DurabilityQosPolicy47
 Requested Offered PresentationQosPolicy65
 Requested Offered ReliabilityQosPolicy68
 RequestedDeadlineMissedStatus87
 RequestedIncompatibleQosStatus88
 ResourceLimitsQosPolicy68
 resume_publications.218
 Return Codes7
 return_loan361
 return_loan (abstract).330

S

sample_state451
 SampleInfo373
 SampleInfo Class451
 SampleLostStatus90

SampleRejectedStatus90
 set_default_datareader_qos303
 set_default_datawriter_qos.219
 set_default_participant_qos161

set_default_publisher_qos	145	State Definitions	456
set_default_subscriber_qos	146	State Masks	456
set_default_topic_qos	147	State Per Sample	452
set_enabled_statuses	105	Status DataReader Statecraft for a Read Communication	466
set_expression_parameters	189, 194	Status Description Per Entity	76
set_listener	148, 182, 220, 240, 304, 331	Status Per Entity	101
set_listener (abstract)	32	StatusCondition Trigger State	472
set_listener (inherited)	264, 363	Struct QosPolicy	33
set_qos	151, 184, 222, 242, 306, 333	Struct SampleInfo	373
set_qos (abstract)	33	Struct Status	76
set_qos (inherited)	264, 363	Subscriber	467
set_query_arguments	396	Subscriber Statecraft for a Read Communication Status	467
set_trigger_value	100	SUBSCRIBER_QOS_DEFAULT	413
Signal Handling	8	SubscriberListener Interface	376
Single Instance instance_state State Chart	454	SubscriberQos	412
Single Instance view_state State Chart	455	Subscription Module	21, 286
Single Sample sample_state State Chart	452	Subscription Type Specific Classes	308
Snapshot	454, 455	SubscriptionMatchStatus	92
SQL Examples	483	suspend_publications	223
SQL Grammar in BNF	481		
SQL Token Expression	482		

T

take	364, 458	take_w_condition (abstract)	336
take (abstract)	334	Thread Safety	8
take_instance	365, 459	TimeBasedFilterQosPolicy	71
take_instance (abstract)	335	Topic Definition Example	475
take_next_instance	367	TOPIC_QOS_DEFAULT	415
take_next_instance (abstract)	335	TopicDataQosPolicy	72
take_next_instance_w_condition	369	Topic-Definition Module	18, 173
take_next_instance_w_condition (abstract)	336	Topic-Definition Type Specific Classes	197
take_next_sample	370, 458	TopicListener interface	195
take_next_sample (abstract)	336	TopicQos	413
take_w_condition	371, 458	TransportPriorityQosPolicy	72

U

unregister_instance	264	unregister_instance_w_timestamp (abstract)	244
unregister_instance (abstract)	243	UserDataQosPolicy	73
unregister_instance_w_timestamp	268		

V

Var Reference Types	10	view_state	454
-------------------------------	----	----------------------	-----

W

wait	96	write (abstract)	245
wait_for_historical_data	337	write_w_timestamp	272
wait_for_historical_data (inherited)	372	write_w_timestamp (abstract)	245, 246
write	269	WriterDataLifecycleQosPolicy	74

